Received: 22 May 2018 | Revised: 22 February 2020 | Accepted: 28 February 2020

DOI: 10.1002/jnm.2747

EMF18 SPECIAL ISSUE

WILEY

Encoding electromagnetic transformation laws for dimensional reduction

Marcus Christian Lehmann | **Mirsad Hadžiefendić** | **Albert Piwonski** | **Rolf Schuhmann**

Theoretische Elektrotechnik, Technische Universität Berlin, Berlin, Germany

Correspondence
Marcus C. Lehmann, Einsteinufer 17, D-10587 Berlin, Germany.
Email: lehmann@tet.tu-berlin.de

Abstract

Electromagnetic phenomena are mathematically described by solutions of boundary value problems. For exploiting symmetries of these boundary value problems in a way that is offered by techniques of dimensional reduction, it needs to be justified that the derivative in symmetry direction is constant or even vanishing. A generalized notion of symmetry can be defined with different directions at every point in space, as long as it is possible to exhibit unidirectional symmetry in some coordinate representation. This can be achieved, for example, when the symmetry direction is given by the direct construction out of a unidirectional symmetry via a coordinate transformation which poses a demand on the boundary value problem. Coordinate independent formulations of boundary value problems do exist but turning that theory into practice demands a pedantic process of backtranslation to the computational notions. This becomes even more challenging when multiple chained transformations are necessary for propagating a symmetry. We try to fill this gap and present the more general, isolated problems of that translation. Within this contribution, the partial derivative and the corresponding chain rule for multivariate calculus are investigated with respect to their encodability in computational terms. We target the layer above univariate calculus, but below tensor calculus.

KEYWORDS

computational electromagnetism, coordinate transformations, lambda-Calculus

1 | INTRODUCTION

There is a variety of different formulas for the transformation of vector components of fields and fluxes in classical electromagnetism. When changing the coordinate-system, the vector components need to be transformed because vector components quantify directions induced by the coordinate-system. This results in a different matrix-transformation

[Correction added on 21 July 2020, after first online publication: the ENDNOTES links were missing and have since been added in this version.]

scheme, depending on the physical meaning of the vectors in question. Different transformation properties of the objects considered in electromagnetic theories have been known for a long time.[1,2] They can be systematically formulated within tensor calculus at the cost of using antisymmetric tensors. Representing electromagnetic objects with antisymmetric tensors leads to a high amount of combinatorics in tensor calculus, especially when resolving permutations. The theory of differential forms provides a formalism to abstract over that.

Within the domain of computational electromagnetism, there exist several formalisms to represent physical entities with mathematical objects. Most notably are quaternions, vectors, tensors, Clifford numbers, and differential forms.[3] The borders of the theories for these objects, that is, the precise number of logical laws belonging to each theory, are differently blurry.

When representing electromagnetic potentials, fields, fluxes, and densities with *differential forms*, the physical space is modeled by the notion of a *manifold*. An electromagnetic boundary value problem can be posed with the help of an *observer structure*[4] within the theory of differential forms. For a differentiable manifold M[1], a smooth nonzero vector field T on M, and a smooth one form τ on M such that $\tau(T) = 1$, the pair (T, τ) is called an *observer structure*. Using this observer structure (T, τ), the differential operators d_τ and $\mathcal{L}_T$ can be established.

A generic boundary value problem over one domain can be transformed to another domain by transforming the involved differential forms. Boundary value problems are regarded *equivalent*[4] if they can be transformed into each other in that way. If a boundary value problem is suitable for using techniques of dimensional reduction, then these techniques can be applied to all equivalent boundary value problems. For an observer structure (τ, T) on a manifold M, an observer structure (γ, Γ) on a manifold N, on these entities a transformation $F : N \to M$ and on the differential forms the induced transformation F^*, the two formulations of Figure 1 pose the same boundary value problem. There is a very regular pattern present in these equations stating what needs to be done to transform a boundary value problem: the differential forms of the original boundary value problem have to be transformed with the *pullback* F^* to appear in the transformed boundary value problem. An introduction into the calculus on manifolds and an introduction of differential forms can be found in the literature.[5]

On a machine, computations for solving a boundary value problem operate on *number data*—the numbers that are stored within the machine's memory. In the current formulation, it might not be that obvious anymore how to convert the original number data into the number data for the transformed boundary value problem in terms of actual computations. A confident implementation of a program benefits from an obvious description of the computation. Therefore, multiple formulations complement each other: for the computation, low-level matrix-operations and index-operations can directly be executed by the machine, but for deriving the computation, only the high-level differential forms statements can be overviewed. We are convinced that high-level abstractions as in Figure 1 pay off in the most beneficial way only, when stacked on top of a layer providing

(a) a good abstraction to provide coordinate transformation rules in terms of matrix-based or just general computation schemes for a given tensorial formulation and
(b) a good abstraction to incorporate combinatorial notions, especially the enumeration of permutations, which enables the reasoning on a level of differential forms to be automatically transferred into a tensorial representation.

The purpose of an implementation is to put the machine into a state that is most efficient for processing all necessary computations of a numerical scheme, which solves the boundary value problem. Abstractions help in organizing the implementation but should not prevent to use the machine in its most efficient way. Therefore, most abstractions are usually stripped before a cost-intensive computational task is started. They should only allow to produce an efficient computational scheme on spot in some form that is available on the machine: matrix or parallel or other kinds of efficient computational primitives. It is important to emphasize that the corresponding raw number data do not need to change for every transformation process.

$$
\begin{aligned}
d_\tau E &= -\mathcal{L}_T B \\
d_\tau H &= J + \mathcal{L}_T D \\
d_\tau D &= \rho \\
d_\tau B &= 0
\end{aligned}
\quad \Leftrightarrow \quad
\begin{aligned}
d_\gamma F^* E &= -\mathcal{L}_\Gamma F^* B \\
d_\gamma F^* H &= F^* J + \mathcal{L}_\Gamma F^* D \\
d_\gamma F^* D &= F^* \rho \\
d_\gamma F^* B &= 0
\end{aligned}
$$

FIGURE 1 Formulation of a generic electromagnetic boundary value problem[4] on the manifold M with differential forms (left) and an equivalent boundary value problem on the manifold N (right). Material laws and boundary conditions are omitted but they also follow the same pattern of transformation

For the first part (a), that is, the generation of transformation rules, in this article, we show a way to realize such a layer that is independent of the actual function representation. The second part (b) is motivated in Section 2.5 and not treated in this article.

The rest of this article is organized as follows: Since this is an interdisciplinary topic, we will give some necessary context for readers from different domains. This context is tailored to an implementation on a machine. In Section 1.1, we introduce the notion of the *frame bundle* and *associated bundles* to the frame bundle from *bundle theory* in order to define *geometric quantities* and give the general transformation rule for (p, q)-tensor ω-densities. In Section 1.2, we introduce shape functions and degrees of freedom of the finite element method in the context of differential forms. An introduction to the untyped λ-calculus is given in Section 1.3. A definition of the partial derivative in terms of a univariate derivative is given in Section 2. In Section 3, the untyped lambda calculus is augmented with axioms for a typed variant for the purpose of expressing the calculations of the previous section. In Section 4, we give a guideline on how this augmented lambda calculus can be applied in a software project.

1.1 | Electromagnetical context

Electromagnetic theory is concerned about the spatial and temporal relation of different physical quantities such as potentials, forces, fluxes, and densities.[6] These are often grasped with respect to a coordinate system and its *coordinate-induced directions*. A base for all directions at a point is called a *frame*.[7] The coordinate system is called a *chart* and it is modeled as a continuous mapping from *points p* of a topological space to *their* coordinates within $\mathbb{R}^n$. A collection of such systems is called an *atlas* and if the whole space can be covered by overlapping charts into $\mathbb{R}^n$ for the same n, it is called *locally euclidean*. A *topological manifold* is defined by additionally demanding the *Hausdorff*[4] property and the space being *second countable*.[4] If chart transition functions of an atlas are arbitrarily often differentiable, the atlas is called *smooth*. Two atlases over M are *smoothly equivalent* when their union is also a smooth atlas over M. An equivalence class of smoothly equivalent atlases over M is called a *smooth structure*. A topological manifold M endowed with a smooth structure is called a *smooth manifold*. Analogously, regarding only k-times differentiable chart transition functions for $k > 0$ leads to a *differentiable structure*. A topological manifold endowed with a differentiable structure is called a *differentiable manifold*.[4] On the differentiable manifold, we defined the observer structure that was necessary to establish the differential operators for the boundary value problem in the Introduction. A far-reaching introduction on this topic can be found in the literature[7,8]. Having two manifolds, one called *total space* E and one called base space M, and a continuous surjective function $\pi : E \to M$, the tuple (E, π, M) is called a *bundle* of manifolds. Here the preimage $\text{preim}_\pi(p)$ of a point $p \in M$ with respect to π is called *fiber at the point p*, denoted by F_p. If fibers of all points are homeomorphic to some manifold F, the bundle is called a *fiber bundle with typical fiber F*. Globally over the manifold, tensor *fields*, vector *fields*, and differential forms are considered. They are modeled as sections of some fiber bundle where locally at a point tensors, vectors and covectors of some algebra are considered. One specifically important algebra for that purpose is the *exterior algebra* of local covectors from global differential forms.

When answering "why exterior differential forms" are useful as a formalism for the modeling of electromagnetic laws, some authors[7] justify this with "the alternating algebraic structure of integrands that gave rise to the development of exterior algebra and calculus which is becoming more and more recognized as a powerful tool in mathematical physics"[2]. Furthermore, we will make use of the generalization of a tensor, the *geometric quantity*, being "defined by the action of the general linear group on a certain set of elements"[3]. Examples are tensor-valued differential forms and twisted tensors.

Electromagnetism as a physical effect does not depend on a chosen coordinate system, which— as a property— is called *general covariance*. In our new wording, a geometric quantity at some point should not depend on a chosen frame. Now, a standard mathematical conjuring trick in order to avoid an arbitrary choice, is to attach that choice to the objects in question—to attach the chosen frame to the quantity in our case.

The theory of associated fiber bundles can describe different kinds of fiber bundles that fulfill an equivalence relation. This equivalence relation is expressed by the means of a Lie group G with respect to some G-bundle. A G-bundle will be introduced straightaway.

For a manifold M, a cover $\{U_\alpha\}$ of M by open sets, a vector space V and a group representation $\rho : G \to GL(V)$ of G on V, where $GL(V)$ is the general linear group over V, it is possible to obtain[8] a vector bundle (E, π, M) using transition functions $g_{\alpha\beta} : U_\alpha \cap U_\beta \to G$. This is only possible when the *compatibility conditions*[8]

$$g_{\alpha\alpha} = 1 \qquad \text{on } U_\alpha$$
$$g_{\alpha\beta} g_{\beta\gamma} g_{\gamma\alpha} = 1 \qquad \text{on } U_\alpha \cap U_\beta \cap U_\gamma$$

are fulfilled. It is done[4] by partitioning the disjoint union $\cup_\alpha (U_\alpha \times V)$ with the equivalence relation

$$(p, v) \sim (p, v') \;\Leftrightarrow\; v = \rho\Big(g_{\alpha\beta}(p)\Big)\; v'$$

to obtain the base space $E = \cup_\alpha (U_\alpha \times V)/\sim$. A vector bundle obtained in this way is called a G-bundle and F is the *standard fiber*.[8] When $V = G$, the G-bundle is called *principal*.[9] The frame bundle LM is a principal G-bundle, where G is the general linear group.

The fiber bundle of frames over a smooth manifold M is called *frame bundle* and denoted by LM. At a point p for some chosen frame $e \in L_pM$ and some geometric quantity $f \in F_p$ from the fiber F_p, we regard the tuple (e, f) as one representation of a geometric quantity at that point. However, one could choose another frame e' which is done by choosing another chart, or coordinate system, and then the new frame is related $\sim$ to the old one by means of the Jacobian J at that point p:

$$(e, f)_p \sim \left(e \triangleleft \mathrm{J}, \mathrm{J}^{-1} \triangleright \mathrm{f}\right).$$

Here $\triangleleft$ is a right action on the frames

$$\left(\vec{e}_1, \ldots, \vec{e}_d\right) \triangleleft \mathrm{J} := \left(\mathrm{J}_1^{\mathrm{m}} \vec{\mathrm{e}}_{\mathrm{m}}, \ldots, \mathrm{J}_{\mathrm{d}}^{\mathrm{m}} \vec{\mathrm{e}}_{\mathrm{m}}\right)$$

and $\triangleright$ is a left action on the fiber

$$\left(J^{-1} \triangleright \mathrm{f}\right)^{i_1 \ldots i_p}{}_{j_1 \ldots j_q} := (\det J)^\omega f^{k_1 \ldots k_p}{}_{l_1 \ldots l_q} \left(J^{-1}\right)^{i_1}{}_{k_1} \ldots \left(J^{-1}\right)^{i_p}{}_{k_q} J^{l_1}{}_{j_1} \ldots J^{l_p}{}_{j_q} \tag{1}$$

for all, so-called (p, q)-tensor ω-densities. Both definitions make use of a *sum convention*, summing over all equal indices.

The Jacobian J is an element of the general linear group from the definition of a geometric quantity. That makes (p, q)-tensor ω-densities a special case of geometric quantities. Not one tuple (e, f) with one chosen frame, but the equivalence class of all tuples that can be related $\sim$ to each other with some J makes a *value* of a geometric quantity at a point p. This is expressed in the inner part of Figure 2: by taking the space $LM \times F$ but partitioning it $(LM \times F)/G$ with respect to a group G. This group is the general linear group in our case. The new fiber bundle $(LM \times F)/G$ is called to be *associated* to the *principal* G-bundle LM.

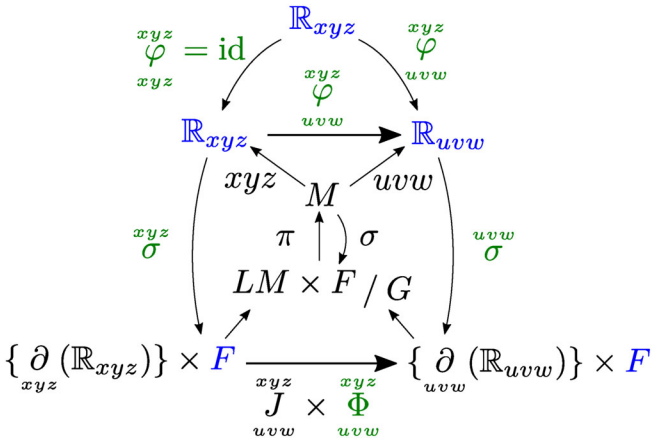

FIGURE 2 Objects involved in a *covariant* treatment of associated fiber bundle. All arrows in this diagram denote functions. The name of the function is written next to its arrow. At the beginning of an arrow is the domain space and at the end of an arrow is the codomain space of the corresponding function. A product of spaces is denoted by $\times$ and similarly the parallel composition of two functions operating on these product spaces is also denoted with $\times$

For the implementation on a machine, we are most likely not to work with a point $p \in M$ but with its coordinates $xyz(p) \in \mathbb{R}^3$. These coordinates are with respect to some chart $xyz : M \to \mathbb{R}^3$, where $\mathbb{R}^3$ is denoted as $\mathbb{R}_{xyz}$ in Figure 2. That chart xyz induces directions and in particular one concrete frame $\underset{xyz}{\partial}\left(\mathbb{R}_{xyz}\right)$ at each point p. To this frame, there corresponds exactly one number data from the fiber F so as programmers we consider the *chart representation* $\overset{xyz}{\sigma}$ of a section σ in the implementation. When changing charts, the chart transition map $\underset{uvw}{\overset{xyz}{\varphi}}$ transforming the coordinates has an analog $\underset{uvw}{\overset{xyz}{J}} \times \underset{uvw}{\overset{xyz}{\Phi}}$ operating on the field representations. Generally we have

$$\Phi(f) \coloneqq J^{-1} \triangleright \mathrm{f}.$$

You can find an interpreted version of Figure 2 in Section 4 as Figure 8.

Usually, one only represents the blue bits of Figure 2 as data and the green bits of Figure 2 as computations in an implementation. The remaining black bits might be treated in an *opaque* way. This is a technique in creating programming interfaces where objects are exposed via references, which are of a defined reference type. That reference is called *opaque* when referring to unexposed or even undefined data while the representation of the reference itself is known[5]. Even though the black bits are not themselves represented in an implementation as number data, their rules of operation are a candidate for entering an implementation as rules of opaque references. Opaque references can be used to restrict the usage of operations on number data to valid cases. The amount and flexibility of expressible restrictions for that purpose is a property of the targeted programming language. In Section 3, we make use of a *dependently typed*[10] programming language that offers high flexibility in expressing restrictions to increase confidence in our approach. In our performance critical code we make use of a *deterministic just-in-time-compiled* programming language that offers partial recompilation and high flexibility for code-generation to help putting the machine into its most efficient state for a computation.

The theory of associated fiber bundles of the frame bundle LM provides a notion of (p, q)-tensor ω-density. This notion is sufficient to express all electromagnetic quantities of interest and they share a single transformation law (1). The transformation F^* of Figure 1 used to transform an electromagnetic boundary value problem follows the rules of Φ from Figure 2: having a single explicit definition for all the various (p, q, ω) transformations, makes this theory very promising as a starting point for an implementation in a software project. Furthermore, it is observable that such software will heavily rely on the correct evaluation of Jacobians at the right coordinates of possibly chained transformations. That is the reason why we are so interested in a very solid foundation of encoding partial derivatives and their chain rule in Sections 2 and 3.

1.2 | Software context

There is much within computational electromagnetism that counts as *software*. This community has a history of incorporating guidance from *mathematical structure* of the electromagnetic theory into the development of *consistent* and *stable* numerical methods.

When speaking of numerical software, this article focuses on the finite element method, which is a *Galerkin method* that can be expressed in terms of the *finite element exterior calculus*[11]. There are a lot of common mistakes leading to wrong solutions of the finite element method. The reason of failure is often not that obvious.[11] This nonobviousness is mirrored in an extensive development of Galerkin methods, and in particular the finite element method, within past decades.

For a *numerical* consideration, that is, for the purpose of establishing proven guarantees of certain *errors*, a numerical method is abstractly modeled using an abstract Hilbert space V. It is assumed that the numerical problem can be expressed using a bounded bilinear form $B : V \times V \to \mathbb{R}$ and a bounded linear form $F : V \to \mathbb{R}$ as

$$\text{find } u \in V \text{ such that } \forall\, v \in V \;.\; B(u,v) = F(v). \tag{2}$$

The problem is called *well-posed* if an unique solution u exists and the *solution mapping* $F \mapsto u$ is bounded again. Using that formulation, a Galerkin method is characterized by a family of *finite dimensional, normed* spaces V_h indexed by parameter h that in *some sense* approximate V. The Galerkin method for that family of spaces V_h, a bilinear form $B_h : V_h \times V_h \to \mathbb{R}$ and a linear form $F_h : V_h \to \mathbb{R}$ poses

$$\text{find } u_h \in V_h \text{ such that } \forall\, v \in V_h \;.\; B_h(u_h,v) = F_h(v). \tag{3}$$

It is desired to prove that the property of "V_h approximating in *some sense*[6] V as h *advances*" is conveyed to "u_h approximating in *some sense* u as h advances." The *finite* element method is a Galerkin method where elements of the basis of V_h have *finite* support, that is, they are nonzero only on a small part of the considered domain.

A construction of bases for a family V_h of spaces can be proven to be consistent and stable when used in a Galerkin method. The finite element exterior calculus provides constructions of classes of finite element bases whose Galerkin methods were proven to be consistent and stable.[11,12] This was done utilizing notions from differential geometry and algebraic topology in order to develop methods for error analysis. It is necessary to do this within a functional analytic setting because a notion of *approximating* and *error* and therefore *consistency* and *stability* of a numerical method do ultimately origin here.

We consider the explicit construction[12] of two families of explicit local bases. Here the approach was "not trying to find hierarchical bases, but rather [...] generalize the explicit Bernstein basis".[12] Where it is easy to give a *spanning set* of polynomials with meeting requirements, it is much harder[12] to provide a basis of *linearly independent* polynomials.

A key insight is to decompose this construction of base elements and define the polynomial base in terms of smaller *shape functions*. Multiple adjacent of those shape functions are *recombined* into one base element by enforcing *proper interelement continuity* conditions.[12] This approach is sometimes called an *assembly*[12][7]. With its interelement continuity conditions, this process is the reason why multiple shape functions from adjacent pieces of a domain share the same *degree of freedom*. The presented[12] assembly process for the construction of basis functions, is "a straightforward consequence of the geometric decomposition of the finite element spaces".[12]

For both families of shape function spaces for each *simplex* T and each subsimplex, sometimes called *face*, f with $r \geq 1$ and $0 \leq k \leq d$ and $d = \dim f \geq k$, there is a shape function space and there are *degrees of freedom*. One is the shape function space of polynomial differential forms $\mathcal{P}_r\Lambda^k(T)$ with corresponding degrees of freedom

$$u \mapsto \int_f (\operatorname{tr}_f u) \wedge q \quad : \quad \mathcal{P}_r\Lambda^k(T) \to \mathbb{R},$$

where q is from $\mathcal{P}^-_{r+k-d}\Lambda^{d-k}(f)$ and $\wedge$ is the exterior product. For differential forms, the trace operation tr is the pullback $\imath^*_{f,T}$ of the inclusion $\imath_{f,T}: f \to T$

$$\operatorname{tr}_f = \imath^*_{f,T}.$$

The other one is the shape function space of polynomial differential forms $\mathcal{P}^-_r\Lambda^k(T)$ with corresponding degrees of freedom given by

$$u \mapsto \int_f (\operatorname{tr}_f u) \wedge q \quad : \quad \mathcal{P}^-_r\Lambda^k(T) \to \mathbb{R},$$

where q is from $\mathcal{P}_{r+k-d-1}\Lambda^{d-k}(f)$.

The construction of the base elements for a shape function space is "somehow a complicated business"[8] and provided[12] in terms of:

- a simplicial complex
- taking the set of subsimplices of a given simplex
- restriction maps from a simplex to one of its subsimplices and inclusion maps the other way around
- barycentric coordinates
- the exterior derivative of barycentric coordinates
- piecewise polynomial differential forms
- the pullback of polynomial differential forms along a restriction map
- multi-indices
- the index-set associated with a face of the simplicial complex
- the set of all order preserving maps of indices
- taking the *support* of a multi-index
- taking the *range* of an order preserving map.

For every item on this list, we will probably have some correspondence within an implementation for a machine. A simplicial complex is usually given by a *mesh*. It is mostly stored in two separate parts. One part is

an abstract simplicial complex consisting just of the combinatorial information, which is sometimes called the *mesh topology*. The other part is additional data, which can be used to create homeomorphisms from the standard simplices to the given ones. This data form a *parametrization* and provide barycentric coordinates. In the case of a simplicial complex, this data might just contain vertex coordinates, but it becomes more interesting for *curved* cells.

Multi-indices and order-preserving index maps are rooted in the combinatorial domain. Their representation in an implementation might be exploited in a clever way. The most intriguing correspondence we think is the one of polynomial differential forms and their pullbacks. These can be resolved within a pen and paper computation[9] and, then, the resulting polynomials can be implemented very carefully. However, it also seems reasonable to formulate the whole construction of a shape function element within a programming language. One of our goals within this article is investigating how to do so in an appropriate way.

This approach essentially lifts the implementation to a metalevel. Previously, as programmers, we were seeking an implementation to perform a numerical computation in the most efficient way for a given machine. Now, we have to program an implementation that is able to produce another, more concrete, implementation, which in turn is able to perform the numerical computation in the most efficient way for a given machine. The efficiency of this metaimplementation is usually not critical for the efficiency of the resulting implementation.

One obvious technique is to generate source code of an implementation with the metaimplementation. The programming language of the metaimplementation does not need to be the same as the programming language for the targeted implementation. Most programming languages offer metaprogramming constructs to generate computations and data structures and the expression of constraints that are to be checked during this generation. These constraints are used to restrict the argument's domain of a metacomputation. The templating system[13] of the C++ programming language is a very popular choice in the community of computational electromagnetism.[14] This might be partly because it allows to use the same language for the implementation and the metaimplementation. While this choice of programming language helps the programmer in putting the machine into its most efficient state, it offers limited flexibility in expressing logical constraints for the valid application of metacomputations. Therefore, the expression of algebraic rules from a construction of finite element bases might only be partially incorporated. When seeking for confidence, it is critical to be able to express all rules that are needed to be confident of. These rules are expressed in the programming language of the metaimplementation in order to have them checked automatically. We might even claim that the usefulness of checking rules critically depends on the completeness, or coverage, of the rules regarding *all possible cases*. Putting it in another way: we claim that

- achieving high efficiency is the biggest challenge when programming the implementation, whereas
- achieving high validity is the biggest challenge when programming the metaimplementation.

That is why we advocate the use of a programming language with a checking mechanism for *dependently typed*[10] expressions to formulate constraints. For the metaprogram, this offers a chance to express all algebraic rules completely. This is relevant because metaimplementation techniques seem to become more and more unavoidable in modern high-performance computing.

1.3 | Computational context

Within this contribution, the partial derivative and its corresponding chain rule for multivariate functions will be investigated with respect to their encodability in *computational terms*. A functional analytic setting is very powerful for an analysis of problems related to partial differential equations. In this article, we will treat the operation of taking the derivative of a univariate function in a more *synthetic* way. The derivative operation will be embedded into a more general context of computation, where some basic properties become assumptions of that embedded derivative operation.

In this article, the understanding of computational terms is backed by lambda-calculus (λ-calculus), which serves as a model, or definition, of *effectively calculable* functions. That calculus was originally developed by Church in 1936[15] and we will follow a modern treatise[16] of the resulting findings. We will take a *type-free*[10] λ-calculus that is extended in Section 3 to a typed variant. The type-free λ-calculus is constituted by a set Λ of λ-terms built up from an infinite set of *variables* $V = \{v, v', v'', \ldots\}$ using *application* and *function abstraction*:

$$\begin{aligned} x \in V &\Rightarrow x \in \Lambda, \\ M, N \in \Lambda &\Rightarrow (M\ N) \in \Lambda, \\ M \in \Lambda, x \in V &\Rightarrow (\lambda\ x\ M) \in \Lambda. \end{aligned}$$

We choose the convention to suppress the outermost parenthesis in $(\lambda\ x\ M)$ when it is unambiguous and to add a separating dot inbetween x and M, resulting in $\lambda\ x\ .\ M$. On these terms, an operation of *substituting* N for the free occurrences of x in M can be defined and is denoted by $M[x := N]$.

Furthermore, there are binary relations for η-reduction, α-conversion, and β-reduction reading from left to right:

$$\lambda\ x\ .\ f\ x \equiv_\eta f \tag{4}$$

$$\lambda\ x\ .\ f\ x \equiv_\alpha \lambda\ y\ .\ f\ y \tag{5}$$

$$(\lambda\ y\ .\ M)\ x \equiv_\beta M[y := x]. \tag{6}$$

We will regard two terms as computationally equivalent if they can be related to each other in these ways. Therefore, a symmetric $\equiv$-symbol is already present here, although η-reduction, α-conversion, and β-reduction are defined as operations from the left-hand side to the right-hand side in the previous listing. The usage of λ-calculus will be elaborated in more detail in Section 2 and put in a more rigorous setting in Section 3.

An introductory survey, reaching out to the techniques[11] used in Section 3, can be found in the literature[17] as the *propositions as types* paradigm. This paradigm pictures the development from λ-calculus to the *proofs-as-programs* and *propositions-as-types* interpretation through one of the most prominent developments within theoretical computer science: the Curry-Howard correspondence.

The rules of the black bits of Figure 2, mentioned in Section 1.1, lead to constraints for restricting an argument's domain of a metacomputation, mentioned in Section 1.2. These rules can be formulated as propositions of objects within the theory of differential forms. Analogously, the constraints for restricting an argument's domain of a metacomputation can be formulated as propositions of an object in the programming language. A compiled programming language is able to perform compile-time checks based on type-equations formulated in the programming language's type system. Expressing a proposition as type-equations, and having these type-equations checked, or affirmed, conveys the affirmation of the type-equations to an affirmation of the proposition. Therefore, using the type system of a programming language for the purpose of establishing validity of a metaimplementation depends on the translatability of propositions into equivalent type equations—equivalent with respect to affirmation. This orientation toward translatability and high validity differs from the algorithmically oriented approach of computer algebra systems.

To support the theory in Section 3, we have used a programming language that is developed precisely for the purpose of establishing a translatability of propositions into equivalent type equations. This choice seems to offer the best chance of being able to express all algebraic rules of a construction of finite element shape functions completely. However, that is an outlook. In this article, we propose an encoding of the partial derivative in λ-calculus as a foundation of a system of algebraic rules. That foundation is tailored toward an application in numerical methods, especially the finite element method.

2 | PROBLEM AND APPLICATION

2.1 | Varying syntax and semantics

We introduced lambda calculus in the standard notation, which is also the notation we use in our implementation later on. However, for this *modeling* part, we switch to a barred-arrow $\mapsto$ notation since it resembles the standard notation of the electromagnetic theory.

References we give here for notation, might be a bit *picked*, and it is, of course, a matter of taste. However, it is this notation that illustrates day-to-day problems when working within the electromagnetic theory.

In one reference[18] from the domain of computational electromagnetism, Bossavit argues[12] about a notation for functions. The given argument is to advertise using an arrow-symbol[13] in order to better emphasize a distinction of functions and *expressions*. It is recommended to denote a function f of the expression $x^2 + 2x + 1$ as

$$f = x \to x^2 + 2x + 1$$

or rather

$$f(x) \coloneqq x^2 + 2x + 1. \tag{7}$$

Where it is stressed that using just = in the second case, could be interpreted as an equality instead of a definition. This is accompanied with an example of a differential operator, helping to resolve some "ambiguity as to which gradient, with respect to x or to y, we mean," making *x the parameter* and *y the variable*, both of which are vectors:

$$\operatorname{grad}\left(y \to \frac{1}{|x-y|}\right) = y \to \frac{x-y}{|x-y|^3}.$$

Those differential operators act on function objects, and their notation might be borrowed from the notation of *higher order functions* in programming. A reference to programming, and especially to λ-calculus is already drawn in that reference.

In the same way that higher order functions, or functional programming in general, are known to have some steep learning curve to overcome, similar applies here. This might be why usually in engineering a *codomain-focused style of notation*, as in (7), is preferred.

We think that some confusion arises by taking expressions and not functions as the dominant objects in calculus. For instance, the Mathematica[19] programming language follows an expression focused approach.

Speaking about expressions, coincidentally also another example[20] is given by Martin-Löf, although he was not up for differential calculus and used it as a mere example for *forms of expressions*:

> The expressions [...] are formed from variables
>
> $$x, y, z$$
>
> by means of various forms of expression
>
> $$(F\ x_1, ..., x_n)(a_1, ..., a_m).$$
>
> In an expression of such a form, not all of the variables x_1, ..., x_n need become bound in all of the parts a_1, ..., a_m. Thus, for each form of expression, it must be laid down what variables become bound in what parts. For example,
>
> $$\int_a^b f \, \mathrm{d}x$$
>
> is a form of expression $(I\ x)(a, b, f)$ with $m = 3$ and $n = 1$, which binds all free occurrences of the single variable x in the third part f, and
>
> $$\frac{\mathrm{d}f}{\mathrm{d}x}(a)$$
>
> is a form of expression $(D\ x)(a, f)$, with $m = 2$ and $n = 1$, which binds all free occurrences of the variable x in the second part f.

Furthermore, Spivak[5] is denoting the multivariate and univariate chain rule by

$$D(g \circ f)(a) = Dg(\ f(a)) \circ Df(a)$$
$$(g \circ f)'(a) = g'(\ f(a)) \cdot f'(a)$$

and introduces the ith partial derivative of f at a as $D_i f(a)$. He alludes that the partial derivative is the ordinary derivative of a certain function, for example, if $g(x) = f(a^1, ..., x, ..., a^n)$ then

$$D_i f(a) = g'(a^i). \tag{8}$$

Notation is briefly discussed and it is mentioned that $D_1 f(u, v, w)$ resolves the usage of a notation like

$$\left.\frac{\partial f(x,y,z)}{\partial x}\right|_{(x,y,z)=(u,v,w)} \text{ or } \frac{\partial f(x,y,z)}{\partial x}(u,v,w).$$

Another issue is framed by Tonti who also dedicates a chapter[6] to revise terminology. There *many kinds of equality* are illuminated. He proposes five different such equalities to be suitable for the purpose of explaining electrodynamics instead of just using a single = for all of them:

$\stackrel{\text{def}}{=}$ definition	$H \stackrel{\text{def}}{=} U + p\,V$, definition of enthalpy
$\equiv$ identity	$a^2 - b^2 \equiv (a+b)(a-b)$, for all a and b
$=$ equation	$3x^2 - 2x = 5$, the variable x is unknown
$\stackrel{\text{mat}}{=}$ material law	$V \stackrel{\text{mat}}{=} R\,I$, Ohm′slaw
$\stackrel{\text{law}}{=}$ general law	$\partial_t \rho + \operatorname{div} \mathbf{J} \stackrel{\text{law}}{=} 0$, conservation law.

This issue could also be summarized by arguing that "the fragment of mathematical symbolese available to most calculus students has only one verb, '='".[21] "That's why students use it when they're in need of a verb."[21] In general, there is "a list of different ways of thinking about or conceiving of the derivative"[21] of a function instead of a single way to do so. These all make their appearance at some point when studying electromagnetism.

We might summarize that these authors propose an expressive notation for what kind of statement is expressed by =, maybe even which kind of objects it relates, and how the variables of an expression are quantified.

Rather than giving a meaning of what *the* univariate derivative is, we treat it synthetically and collect the few properties necessary for introducing a partial derivative on top. In order to resolve the various notations, we have chosen to resemble λ-calculus.

To support multiple interpretations, we chose to explain our usage of λ-calculus with a changed notation. From our experience this better resembles day-to-day notation in computational electromagnetism but is close enough to follow notation of a formalization later-on in Section 3. This choice is made to support readers that are not immediately implementing such λ-calculus but still want to gain some insights about the partial derivative.

2.2 | Yet another notation for functions

Our aim is to connect more high-level theories, such as tensor calculus and differential forms to more low-level theories, such as multivariate calculus and λ-calculus. With tensors and differential forms, it is possible in a tractable way to express sound notions of invariant properties and differentials. In multivariate calculus and λ-calculus, it is possible in a tractable way to express sound notions of an univariate derivative and computations. After such a connection is made, representations and implementations that arguably behave in a way respecting these notions need to be given. Doing so should contribute to the discussion about how higher level *representations of physical entities* can be encoded in a program.

We start with the assumption of a given univariate derivative operation $'$ that for a given univariate function representation f can compute the univariate function representation of the derivative of that function f'. For the computational description, we make use of an untyped, simplified λ-calculus as introduced in Section 1.3. Instead of $\lambda x.\, fx$, we denote function abstraction by $x \mapsto f(x)$ to better resemble day-to-day notation. We emphasize that only the following

rules are used and it does not matter if you do not know λ-calculus yet, if you can familiarize yourself with these four *computational equivalences* (9–12) that are already in use in engineering mathematics and denoted by $\equiv$ here. The meaning of these equivalences is explained in the following. They display as:

$$f \equiv_\eta x \mapsto f(x) \tag{9}$$

$$x \mapsto f(x) \equiv_\alpha y \mapsto f(y) \tag{10}$$

$$(y \mapsto \mathbf{term})(x) \equiv_\beta \mathbf{term}[y{:=}x] \tag{11}$$

$$f(g(x)) \equiv_\circ (f \circ g)(x). \tag{12}$$

The intention of stating these rules is to be able to distinguish and name them. Our application of the η-equivalence on univariate functions (9) states that a function f and the λ-abstraction[14] immediately applying the argument $x \mapsto f(x)$ are computationally equivalent and therefore can be substituted against each other respecting the computation's result. The α-equivalence (10) in this case states, that it does not matter for the computation how the argument is named, of course. Hence, every time $\equiv_\alpha$ appears, the left-hand side can be transformed in a computationally equivalent way to the right-hand side by argument-renaming and vice versa. The β-equivalence (11) expresses that an application of function $y \mapsto \mathbf{term}$, that is, the **term** regarded as dependent on its variable y, to the argument x is computationally equivalent to a $\mathbf{term}[y := x]$, where all occurrences of y are substituted for x. This is denoted by the substitution $[y := x]$ acting on the **term** as a postfix operation. Finally, not that much a rule of λ-calculus but more a definition of the composition operation $\circ$ is the rule (12).

These rules (9–12) are somewhat standard rules that are most likely fulfilled in any context of computation. In λ-calculus every function takes exactly one argument and has one result, which is a perfect interpretation for *univariate* calculus. In computational electromagnetism, the representations of the considered objects, the electric and magnetic fields, the geometry, for example, when given by parametrized coordinates, and coordinate transformations are expressed as *multivariate* functions, taking multiple arguments to multiple results[15]. Multiple arguments can be already thought of being represented as one argument with the help of the notion of a tuple, where the single arguments are separated by commas. Multiple results can be thought of as tuples in a similar manner. Yet, we choose a notation here that allows a multiple-argument-interpretation instead of tuples. It seems most familiar to the engineering community and does not pose a limitation since a multiple-argument-interpretation is translatable to a one-argument-interpretation.

That notation is motivated by the tediousness of multivariate calculus to express function application for these multiple arguments[16]. For a **term**, we denote the *expansion* by **term**..., which should be computationally equivalent to a context where the comma-separation of copies of the term substituted with every single parameter, or variable in our case, of a tuple is applied. If $\underline{x}$ denotes a tuple of four parameters, the expansion of the most simple term, just consisting of $\underline{x}$ itself, corresponds to

$$(\underline{x}...) \equiv_m \left(x^1, x^2, x^3, x^4\right) .$$

Here, the tuple expansion... captures[17] all tuples in the term $\underline{x}$, which is just $\underline{x}$, and expands them to x^1, x^2, x^3, x^4 within the original term to produce the resulting term. The three dots are used frequently in a metalogical manner, where it is clear from the context how to continue the pattern. When it comes to an implementation, one needs to make this pattern-repetition precise. In the following we make use of the three dots... only in the sense of this kind of expansion, where the tuple is again underlined to highlight its meaning as a placeholder. The unexpanded term is denoted in an m-way as computationally equivalent $\equiv_m$ to the expanded one.

The reason for introducing this particular notation is that it supports us in making precise arguments about multivariate functions in the previous sense. Our most important application is to express multivariate function application. For example, suppose g is a multivariate function in $\mathbb{R}^2 \to \mathbb{R}^3$ such that it can be decomposed into functions g^1, g^2, and g^3 in $\mathbb{R}^2 \to \mathbb{R}$, then we have two computationally equivalent terms with the *nested use of the operation of tuple-expansion...*:

$$\left(\underline{g}\left(\underline{x}...\right)...\right)\equiv_m\left(\underline{g}\left(x^1,x^2\right)...\right)$$
$$\equiv_m\left(g^1\left(x^1,x^2\right),g^2\left(x^1,x^2\right),g^3\left(x^1,x^2\right)\right).$$

Here, the green tuple expansion ... captures the green tuple $\underline{g}$, where the blue tuple expansion ... captures the blue tuple $\underline{x}$. Another use[18] is, given that γ is a multivariate function in $\mathbb{R}^1\to\mathbb{R}^3$ that can be decomposed into the functions γ^1, γ^2, and γ^3 in $\mathbb{R}^1\to\mathbb{R}$, then we have two computationally equivalent terms with the expansion ... of *multiple nested tuples* $\underline{\gamma}$ and $\underline{x}$:

$$\left(\underline{\gamma}\left(\underline{x}\right)...\right)\equiv_m\left(\gamma^1\left(x^1\right),\gamma^2\left(x^2\right),\gamma^3\left(x^3\right)\right).$$

Here, the blue tuple expansion... captures both blue tuples $\underline{\gamma}$ and $\underline{x}$. Given the notion of tuples $\underline{x}$, $\underline{y}$ and the operation of tuple-expansion..., we can restate the previous computational equivalences (9–12) in their multivariate version (13–16):

$$f\equiv_\eta(\underline{x}...)\mapsto f(\underline{x}...) \tag{13}$$

$$(\underline{x}...)\mapsto f(\underline{x}...)\equiv_\alpha\left(\underline{y}...\right)\mapsto f\left(\underline{y}...\right) \tag{14}$$

$$\left(\left(\underline{y}...\right)\mapsto\mathbf{term}\right)(\underline{x}...)\equiv_\beta\mathbf{term}\left[\left(\underline{y}:=\underline{x}\right)...\right] \tag{15}$$

$$f\left(\underline{g}(\underline{x}...)...\right)\equiv_\circ(f\circ g)(\underline{x}...)\ . \tag{16}$$

Note especially how expansion interacts with composition of multivariate functions in (16).

One more remark about tuples: you might have noticed that, despite underline and dots, the rules (13–16) exactly match the rules (9–12). That is not a coincidence. Indeed, we could identify a scalar, with the one-tuple of scalars and have just one *generalized* version of the rules. This works for all tuples, including one-tuples, and therefore also for all scalars. Also, in Section 3, we do regard multivariate functions as mapping tuples of scalars to tuples again[19]. Different notation and a reference to parameter-packs are just given, to support an interpretation within a language that does not identify scalars with one-tuples and may even distinguish tuples from a list of function-arguments[20].

You are free to ignore the three dots, when targeting an interpretation where function arguments and tuples are treated the same way[21]. However, as with higher order functions, tuples add a small burden on the learning curve and it is sometimes convenient to just think of a written-out version when comparing with the literature. One can test this preference by looking at $f\ (x, y, z)$ and if that should be *a function f, applied to the function arguments x, y and z*, then the tuple-expansion notation might be a fit. However, if you are comfortable with (x, y, z) being a tuple and if that tuple would be named τ your preferred notation is just $f\ \tau$, then you might want to ignore the dots. When programming, this choice is made by the programming language.

2.3 | Encoding the partial derivative

We make use of the previously introduced equivalences to formulate what a partial derivative should be in that context. It is thought of as being the univariate derivative of a multivariate function, which is regarded as a univariate function only depending on its one argument that we are taking the derivative of. That *univariate regarding of a multivariate function* can be made precise now:

$$h\equiv_\eta(\underline{x}...)\mapsto h(\underline{x}...) \tag{17}$$

$$\equiv_\beta(\underline{x}...)\mapsto\left(x^2\mapsto h(\underline{x}...)\right)\left(x^2\right) \tag{18}$$

$$\equiv_m\left(x^1,x^2,x^3\right)\mapsto\left(x^2\mapsto h\left(x^1,x^2,x^3\right)\right)\left(x^2\right) \tag{19}$$

$$\equiv_\alpha \left(x^1,x^2,x^3\right) \mapsto \left(\mathbf{z} \mapsto h\left(x^1,\mathbf{z},x^3\right)\right)\left(x^2\right) \tag{20}$$

$$\equiv_m \left(\underline{x}...\right) \mapsto \left(\mathbf{z} \mapsto h\left(\underline{x}...\left[\bullet^2 := \mathbf{z}\right]\right)\right)\left(x^2\right). \tag{21}$$

Suppose the multivariate function h is in $\mathbb{R}^3 \to \mathbb{R}$. Then h is computationally equivalent in an η-way to the multivariate function $(\underline{x}...) \mapsto h(\underline{x}...)$ as in (17). Just the inner **term** $h(\underline{x}...)$ of that new multivariate function is computationally equivalent to $(x^2 \mapsto h(\underline{x}...))(x^2)$ in a univariate-β-way (18). To see this, for the example, we look at the expanded version (19). What happened is that the inner abstraction of x^2 is *shadowing*[22] the outer argument x^2. To highlight this difference, we explicitly rename the inner x^2 into an α-equivalent function with $\mathbf{z}$ occurring instead (20). This in a multivariate way constitutes the *substituted expansion of the tuple* $\underline{x}$, denoted as $\underline{x}...[\bullet^2 := \mathbf{z}]$, where entry 2 is replaced with $\mathbf{z}$ as in (21).

This leads to the last rule of computational equivalence that we need for our considerations and it relates a multivariate function application to the use of a univariate function application:

$$\left(\mathbf{z} \mapsto h\left(\underline{x}...\left[\bullet^{\mathbf{i}} := \mathbf{z}\right]\right)\right)\left(x^{\mathbf{i}}\right) \equiv_\beta h(\underline{x}...). \tag{22}$$

To better familiarize with it, looking forward to an implementation, we give the syntax tree of this rule in Figure 3.

That is, finally, enough to define the partial derivative on multivariate functions $f: \mathbb{R}^d \to \mathbb{R}^c$ by the notion of the derivative $'$ on univariate functions. For a general arity and the indices $\mathbf{j} \in [1, c]$ and $\mathbf{i} \in [1, d]$ it is given as the multivariate function

$$\frac{\partial f^{\mathbf{j}}}{\partial x^{\mathbf{i}}} := (\underline{x}...) \mapsto \left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(\underline{x}...\left[\bullet^{\mathbf{i}} := \mathbf{z}\right]\right)\right)'\left(x^{\mathbf{i}}\right), \tag{23}$$

where $f^{\mathbf{j}}$ is the projection $\mathrm{proj}^{\mathbf{j}} \circ f$ of the $\mathbf{j}$th result of the multivariate function f or similarly the $\mathbf{j}$-th part of the decomposition of f in the previously discussed manner.

In that definition (23) we do not use the information about how to name the argument, with respect to which we are taking the partial derivative. That is the case because partial derivatives with differently named arguments are computationally equivalent by α-equivalence:

$$\frac{\partial f^{\mathbf{j}}}{\partial x^{\mathbf{i}}} \equiv_\alpha \frac{\partial f^{\mathbf{j}}}{\partial y^{\mathbf{i}}} \equiv_\alpha \partial_{\mathbf{i}} f^{\mathbf{j}}.$$

There are two remarks here to make. First, the computational equivalence of the partial derivative under renaming of the argument, that is, the α-equivalence, motivates to omit the variable name $\partial_{\mathbf{i}} f^{\mathbf{j}}$. Later-on, however, in the theory of differential forms, this exact spot to give a name to the argument is often used to indicate which charts are involved in the process of coordinate transition[23]. That characteristic results from the use of function-abstraction to express the

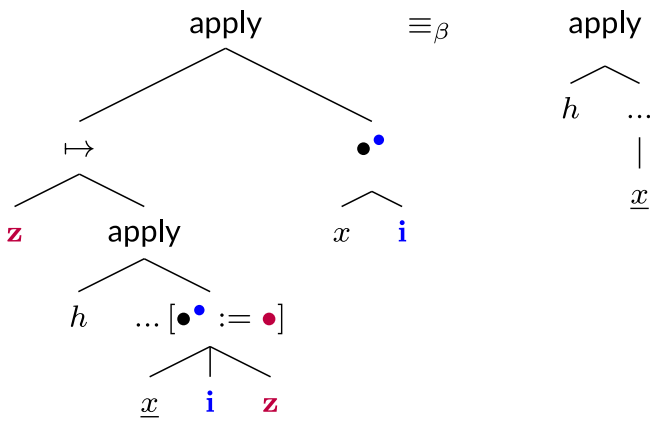

FIGURE 3 Univariate abstraction over an expansion... is expressed explicitly as substituted expansion ...[• := •]to avoid implicit shadowing

partial derivative instead of introducing a new *form of expression* as in the example of Martin-Löf given in Section 2.1. In this way, the definition (23) does not bind any free variables of its argument-terms.

Secondly, for a transition along f from A-coordinates to B-coordinates, that is, where f is a function expressing the B-coordinates in terms of A-coordinates $(\underline{f}(\underline{a}...)...) = (\underline{b}(\underline{a}...)...)$, we have $\frac{\partial f^{\mathbf{j}}}{\partial a^{\mathbf{i}}}(\underline{a}...)$ to constitute the number in the $\mathbf{j}$th row and the $\mathbf{i}$th column of the Jacobi-matrix $\mathbf{J}_f$ evaluated in A-coordinates at $(\underline{a}...)$. That matrix is used to transform the numbers $(\underline{v_B}...)$ that are the vector-components with respect to the B-induced basis at a point given by the same A-coordinates into the numbers $(\underline{v_A}...)$ that are the vector-components with respect to the A-induced basis at the same physical point by matrix–vector-multiplication[24]. This scrutiny forms the foundation of a matrix-translation in terms of the Jacobi-matrix for different kinds of vectors. It is important to gain any support from encoding this logic into the notation and into the program to handle these different calculations and check them for consistency.

2.4 | The chain-rule revised

Using just these established conditions, we will derive what it means to have a notion of a chain rule for the partial derivative, lifting the notion of the univariate chain rule to the multivariate level. The whole calculation is given in Appendix A. In order to create the multivariate listing in Appendix A and the corresponding one for a concrete two-variate case in Appendix B, we have implemented the tuple expansion the previously introduced way.

We begin in (A1) with the partial derivative that can be represented in an implementation not carrying anymore information than written in (A1), i.e. which function $f^{\mathbf{j}} \circ g$ it applies to with respect to which entry $\mathbf{i}$ or directly as the function that we encoded definitionally in (23). In the first case an implementation needs to provide a function that converts these bits of information into that encoding. In the second case we directly operate on these objects. The multivariate function (A2) again does not need more information encoded than written out there and the data structure is very similar to the one resulting from a tree-like encoding of Figure 3. The expanded terms for the two-variate case where $\mathbf{i} = 1$ is given by (B2) and you can follow the expanded variant in Appendix B alongside this investigation.

An equivalent computation (A3) is given by the multivariate $\circ$-equivalence, applying $f^{\mathbf{j}}$ to $\underline{g}$ instead of composing it with g. At this point, we make use of a linearity-property, which needs to be fulfilled for a concrete realization of the univariate derivative $'$ later-on. Namely that the univariate derivative of a multiply occurring argument is given by the sum of the univariate derivatives of each occurrence. We denote this by $=_{\mathrm{lin}'}$ for the two-variate example given by:

$$\begin{aligned} & (\mathbf{z} \mapsto h(\mathbf{z},\mathbf{z}))'(x) \\ =_{\mathrm{lin}'} \; & (\mathbf{z} \mapsto h(\mathbf{z},x))'(x) \\ + \; & (\mathbf{z} \mapsto h(x,\mathbf{z}))'(x)\ . \end{aligned} \tag{24}$$

For our general multivariate notation, h has to be identified with

$$h := (\underline{z}...) \mapsto f\left(\underline{g}(\underline{x}...[\bullet^{\mathbf{i}} := \underline{z}])...\right),$$

leading to the general multivariate variant of this linearity, expressed with a summation $\sum_k$ over a new index k:

$$\begin{aligned} & \left(\mathbf{z} \mapsto f\left(\underline{g}(\underline{x}...[\bullet^{\mathbf{i}} := \mathbf{z}])...\right)\right)'(x^{\mathbf{i}}) \\ =_{\mathrm{lin}'} \; & \sum_k \left(\mathbf{z} \mapsto f\left(\underline{g}(\underline{x}...)...[\bullet^{k} := g^k(\underline{x}...[\bullet^{\mathbf{i}} := \mathbf{z}])]\right)\right)'(x^{\mathbf{i}}), \end{aligned} \tag{25}$$

which expands in the two-variate case for $\mathbf{i} = 1$ to:

$$\begin{aligned} & \left(\mathbf{z} \mapsto f\left(g^1(\mathbf{z},x^2), g^2(\mathbf{z},x^2)\right)\right)'(x^1) \\ =_{\mathrm{lin}'} \; & \left(\mathbf{z} \mapsto f\left(g^1(\mathbf{z},x^2), g^2(x^1,x^2)\right)\right)'(x^1) \\ + \; & \left(\mathbf{z} \mapsto f\left(g^1(x^1,x^2), g^2(\mathbf{z},x^2)\right)\right)'(x^1). \end{aligned}$$

Note the nested substitution in the right-hand-side term of (25) now, where only the application of the k'th decomposition of g is differently applied to the x's of which just the $\mathbf{i}$'th one is replaced with $\mathbf{z}$. Therefore, the linearity $=_{\text{lin}'}$ justifies whether (A4) computes the same result.

$$\begin{aligned}&\left(\mathbf{z}\mapsto f\left(\underline{g}(\underline{x}\ldots)\ldots\left[\bullet^{k}:=g^{k}\left(\underline{x}\ldots\left[\bullet^{\mathbf{i}}:=\mathbf{z}\right]\right)\right]\right)\right)\\ \equiv_{\circ}&\left(\mathbf{z}\mapsto f\left(\underline{g}(\underline{x}\ldots)\ldots\left[\bullet^{k}:=\mathbf{z}\right]\right)\right)\circ\left(\mathbf{z}\mapsto g^{k}\left(\underline{x}\ldots\left[\bullet^{\mathbf{i}}:=\mathbf{z}\right]\right)\right)\end{aligned}\tag{26}$$

The nested substitution is computationally equivalent to the composition of univariate functions containing just a single substitution as in (26) which is the needed transformation that leads to (A5).

At this point, we have encoded the sum of k different univariate derivatives of a composition of two univariate functions (A6), where k-times the univariate chain rule can be applied (A7) to lead to (A8). For the right multiplicand after transforming it in a β-way to the computationally equivalent form in (A9) it matches the definition of the partial derivative on g (A10). The left multiplicand can be turned in a $\circ$- and β-way to the computationally equivalent form (A11-A12) where the definition of the partial derivative again applies. This leads to the common form (A13) of the right-hand side of the chain rule for the partial derivative of the composition of two functions $f^{\mathbf{j}}\circ g$, almost, but not quite:

$$\frac{\partial\left(f^{\mathbf{j}}\circ g\right)}{\partial x^{\mathbf{i}}}=(\underline{x}\ldots)\mapsto\sum_{k}\frac{\partial f^{\mathbf{j}}}{\partial y^{k}}\left(\underline{g}(\underline{x}\ldots)\ldots\right)\cdot\frac{\partial g^{k}}{\partial x^{\mathbf{i}}}(\underline{x}\ldots)\ .\tag{27}$$

The applied calculus enforced an explicit mentioning of the abstraction $(\underline{x}\ldots)\mapsto$ since these are function objects and only if they are applied to the same arguments, the one resulting number is equal for both sides:

$$\frac{\partial\left(f^{\mathbf{j}}\circ g\right)}{\partial x^{\mathbf{i}}}(\underline{x}\ldots)=\sum_{k}\frac{\partial f^{\mathbf{j}}}{\partial y^{k}}\left(\underline{g}(\underline{x}\ldots)\ldots\right)\cdot\frac{\partial g^{k}}{\partial x^{\mathbf{i}}}(\underline{x}\ldots)\ .\tag{28}$$

2.5 | Targeting tensor calculus

In this article our focus is to establish the lower interface that an encoding of the chain rule of multivariate functions demands from an encoding of the univariate chain rule. It was investigated, how to define the partial derivative in computational terms. We have shown in Section 2.4 that this computational context is capable of deriving a chain rule for this definition. In Section 3, we will introduce an augmented λ-calculus based on the requirement to express a derivation of the chain rule from Section 2.4. What remains open for discussion is the question whether that augmented λ-calculus is suitable to express definitions and derivations from tensor calculus. It is also not obvious how the upper interface to tensor calculus should look like. This section motivates why we think that our approach is extendable to express derivations from tensor calculus.

Continuing on (A13), with the $\circ$-equivalence we have a context (A14) where it is possible to make use of a function-level multiplication $\otimes$ that is given by the corresponding point-wise multiplication (A15). This is a binary operation and could be precomposed with a function applying g to the left argument and the identity id to the right argument. Defining such function is in favor for having just one binary operation on the two partial derivatives (A16), making a corresponding data structure definition even more obvious. Establishing a function-level summation $\oplus$ makes it possible to express the chain-rule in a completely so-called *point-free*[25] style (A17). The objects reasoned about in this expression should correspond (denoted by $\cong_T$) to objects of the expression (A18) of tensor calculus, where unfortunately $'$ is a decoration on indices and not to be confused with the univariate derivative. We think that based on the way of that correspondence $\cong_T$ the question of encoding could be answered in a tractable way.

What are the objects of tensor calculus that are common to reason about in computational electromagnetism? In the appendix of his book[6], Tonti collects the notions of:

- tensors and pseudotensors, such as tensor densities and tensor capacities, that differ in their transformation laws on a power of the determinant of the coordinate transition function,

- natural, reciprocal and physical basis vectors, leading to contravariant, covariant and physical components that are number-representations of various kinds of scalars and vectors in electromagnetic theory, and
- algebraic and metric dual vectors that constitute different representations of antisymmetric tensors.

In classical electrodynamics, the physical base is often chosen because of its property to preserve the calculation for the length of a vector. This gives a direct interpretation for the measurement of such a quantity in a cartesian system, which is very valuable in a physical interpretation. These choices are combined with constructions such as the magnetic flux *tuple* of numbers corresponding to the *three-number representation* of the magnetic flux bi-covector at a point and similar constructions. Therefore, we think that it becomes arguable to investigate the computational aspects of such a correspondence. In accordance to follow his notation, which is very well chosen to support the application in various physical theories, we give correspondences in Figure 4.

Note especially, the choice of different symbols λ and Λ to reflect the information in which *logical direction*[26] the partial derivative has to be taken and the drive to name the argument, x or x' respectively, to remember the coordinate transition function's domain. The difference between tensor calculus and the presented formalism is that we regard objects that are functions and function compositions where tensor calculus has a notion of *coordinate system*. That is the key abstraction necessary to use in an implementation suitable of computing the chain-rule as a supporting layer. Consequently, we had no need to name the arguments and it is indeed not possible by α-equivalence $\equiv_\alpha$ to encode that additional information.

Just to oppose it, we give in Figure 4 another popular choice for denoting the partial derivative in tensor calculus $\mathrm{J}^{\mathbf{k}}_{\mathbf{h}'}$ for λ^k_h and $\mathrm{J}^{\mathbf{h}'}_{\mathbf{k}}$ for Λ^h_k. As mentioned before, the $'$ here should not be confused with the univariate derivative. The $'$ is a decoration on the indices $\mathbf{k}$ and $\mathbf{h}$ to represent their *coordinate system belongingness*. Choosing different kinds of decorations for the indices to omit giving indices to the indices is an inevitable problem when multiple coordinate systems are considered. In addition to that choice, there is the legitimate choice of the property of coordinate system belongingness being one of the index or being a property of the *partial derivative object* itself. The former perspective is taken in the notation we opposed which where the latter was denoted λ or Λ respectively. This state of affairs is also shown in Figure 5. An answer to that question of choice highly influences the encoding of tensor calculus expressions for the purpose of an implementation.

As promised in the title, we will show here transformation laws for the magnetic flux B and the electric field E, although the reason of this article is not the result but the process of deriving these laws. For a clarified choice of $\cong_T$, which we did not yet made in this article, suppose that Z, A, and B are given by left decorated $\grave{z}$, undecorated a and right decorated b' coordinate systems. In this notation, for clarification, the coordinate system belongingness is redundantly encoded in the choice of the letter, as well as in the decoration of that letter. This amounts to the habit that in the calculus of multivariate functions, just different letters are used, where in tensor calculus only different decorations are used. Then, for the two transition functions $g : Z \to A$ and $f : A \to B$ the tensor calculus expression that relates

Tonti 2013 $x'^h = f^h(x^k)$	this paper $f : A \to B$
$\lambda^k_h(x') \stackrel{\text{def}}{=} \frac{\partial x^k(x')}{\partial x'^h}$	$\mathrm{J}^{\mathbf{k}}_{\mathbf{h}'}(\underline{a}\ldots) \cong_T \frac{\partial f^{\mathbf{k}}}{\partial a^{\mathbf{h}}}(\underline{a}\ldots)$
$\Lambda^h_k(x) \stackrel{\text{def}}{=} \frac{\partial x'^h(x)}{\partial x^k}$	$\mathrm{J}^{\mathbf{h}'}_{\mathbf{k}}(\underline{b}\ldots) \cong_T \frac{\partial \left(f^{-1}\right)^{\mathbf{h}}}{\partial b^{\mathbf{k}}}(\underline{b}\ldots)$
$\Delta(x') \stackrel{\text{def}}{=} \det(\lambda^k_h(x'))$	(*proposes permutations*)
$g \stackrel{\text{def}}{=} \det(g_{hk})$	(*proposes permutations*)

FIGURE 4 Denotational correspondences, where g_{hk} is the metric tensor and det being the determinant

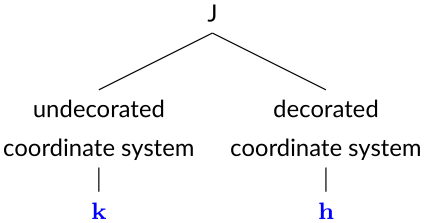

FIGURE 5 Encoding of the partial derivative used in tensor calculus

the covariant components $B_{i'j'}$ of the bi-covector of the right decorated coordinate system b' to the ones B_{ij} of the undecorated coordinate system a is given by:

$$B_{i'j'} = J^{i}_{i'} J^{j}_{j'} B_{ij},$$

where *free* indices are highlighted in blue and *bound* indices, which are summed over, are highlighted in green. This translates into:

$$B_{i'j'} \equiv (\underline{b}...) \mapsto \sum_{i} \sum_{j} \frac{\partial \left(f^{-1} \right)^{i}}{\partial b^{i'}} (\underline{b}...) \cdot \frac{\partial \left(f^{-1} \right)^{j}}{\partial b^{j'}} (\underline{b}...) \cdot B_{ij} \left(\underline{f}(\underline{b}...)... \right).$$

As B_{ij} should be regarded to *naturally* live on the undecorated coordinates a and the resulting object $B_{i'j'}$ to live on the right decorated coordinates b', a precomposition with f is necessary to obtain the B_{ij} value at b' coordinates. Although this transformation goes in the same logical direction as the functions g and f are defined, the partial derivatives of inverses of these functions appear due to the contravariant transformation property of the considered electromagnetic quantity.

Tensor calculus is concerned about the invariance properties of different quantities. Suppose that two Jacobians cancel each other out in the following way

$$J^{i'}_{i} J^{i}_{j'} = \delta^{i'}_{j'},$$

where $\delta^{i'}_{j'}$ is the Kronecker delta, which is 1 for $i' = j'$ and 0 otherwise. Then, it is *easy* to see that the transformations of the tensor components in $S^{ij} T_i U_j$ will cancel each other out. If the independent transformations of S^{ij}, T_i and U_j are

$$S^{ij} = S^{i'j'} J^{i}_{i'} J^{j}_{j'}, \quad T_i = T_{k'} J^{k'}_{i}, \quad \text{and} \quad U_j = U_{l'} J^{l'}_{j},$$

then we have for the composed term

$$\begin{aligned} S^{ij} T_i U_j &= \left(S^{i'j'} J^{i}_{i'} J^{j}_{j'} \right) \left(T_{k'} J^{k'}_{i} \right) \left(U_{l'} J^{l'}_{j} \right) \\ &= S^{i'j'} \left(J^{k'}_{i} J^{i}_{i'} \right) \left(J^{l'}_{j} J^{j}_{j'} \right) T_{k'} U_{l'} \\ &= S^{i'j'} \left(\delta^{k'}_{i'} \right) \left(\delta^{l'}_{j'} \right) T_{k'} U_{l'} \\ &= S^{i'j'} T_{i'} U_{j'}. \end{aligned}$$

Here, the parentheses are present only for clarification since this tensor expression represents scalar multiplications.

In our previous example, the transformation of a bi-covector needs to be *justified* by invariance properties that are expressable within tensor calculus. The appearance of Jacobians is due to this invariance. With our proposed formalism, we can express this as a computational equivalence. If we were to have a representation for $B_{i'j'}$ and B_{ij} then such an equivalence should be derivable at this tensor-calculus-layer. This might motivate that the statements that need to be proven, which arise in tensor calculus, are expressable in our proposed formalism.

Another example is the tensor calculus expression relating the contravariant components $E^{i'}$ of vector E in the right decorated coordinate system b' to the ones E^{i} of the left decorated coordinate system z` is given by:

$$E^{i'} = J^{i'}_{i} J^{i}_{i} E^{i} = J^{i'}_{i} E^{i},$$

which translates into:

$$
\begin{aligned}
E^{\mathbf{i}'} \equiv (\underline{b}\ldots) \mapsto & \sum_{\mathbf{i}} \sum_{\grave{\mathbf{i}}} \frac{\partial f^{\mathbf{i}'}}{\partial a^{\mathbf{i}}} \left(\underline{f}^{-1}(\underline{b}\ldots)\ldots \right) \\
& \cdot \frac{\partial g^{\mathbf{i}}}{\partial z^{\grave{\mathbf{i}}}} \left(\underline{g}^{-1} \left(\underline{f}^{-1}(\underline{b}\ldots)\ldots \right)\ldots \right) \\
& \cdot E^{\grave{\mathbf{i}}} \left(\underline{g}^{-1} \left(\underline{f}^{-1}(\underline{b}\ldots)\ldots \right)\ldots \right). \\
= {}_{\text{chain rule}} (\underline{b}\ldots) \mapsto & \sum_{\grave{\mathbf{i}}} \frac{\partial \left(f^{\mathbf{i}'} \circ g \right)}{\partial z^{\grave{\mathbf{i}}}} \left(\underline{g}^{-1} \left(\underline{f}^{-1}(\underline{b}\ldots)\ldots \right)\ldots \right) \\
& \cdot E^{\grave{\mathbf{i}}} \left(\underline{g}^{-1} \left(\underline{f}^{-1}(\underline{b}\ldots)\ldots \right)\ldots \right)
\end{aligned}
$$

Here again the resulting $E^{\mathbf{i}'}$ should live on the right decorated coordinates b', where the original $E^{\grave{\mathbf{i}}}$ lives on the left-decorated coordinates $\grave{z}$. The transformation happened again in the same logical direction as the functions go, but this time we have transformed twice. To apply the introduced partial differential of the multivariate functions, it becomes necessary to pre-compose with proper inverses to obtain an expression that again depends on the right decorated coordinates b'.

This second example of a chained coordinate transformation makes use of the derived chain rule, which is justified in the current formalism. This should make it easy in an on-top tensor calculus layer to computationally proof

$$
\mathrm{J}^{\mathbf{i}'}_{\mathbf{i}} \mathrm{J}^{\mathbf{i}}_{\grave{\mathbf{i}}} = \mathrm{J}^{\mathbf{i}'}_{\grave{\mathbf{i}}}
$$

when a clarified choice is made how the tensor calculus terms should correspond $\cong_T$ to terms from λ-calculus.

3 | THEORETICAL FRAMEWORK

3.1 | Rules of inference

A logical inference, for example,

$$
\frac{A \qquad B}{A \,\&\, B}
$$

"does not take us from the propositions A and B to the proposition $A \;\&\; B$."[22] "Rather, it takes us from the affirmation of A and the affirmation of B to the affirmation of $A \;\&\; B$."[22] Making this explicit in writing could[27] be.

$$
\frac{\vdash A \qquad \vdash B}{\vdash A \,\&\, B}\,.
$$

there is a need to distinguish two kinds of entities:

- "the entities that the logical operations operate on, which we call propositions"[22] [28], which are "affirmed in an affirmation and denied in a denial."[22]
- and "the things that the logical laws, by which I mean the rules of inference, operate on, which we normally call assertions",[22] which are "those that we prove and that appear as premises and conclusion of a logical inference."[22]

We are examining that topic at this point in the article for two reasons: One is in preparation of stating *introduction rules* for an augmented λ-calculus. The other is because the word *proposition* has a different meaning in logic than it has in most of mathematics.

A logicians issue with the mathematical wording would be that "a theorem is sometimes called a proposition, sometimes a theorem."[22] And thus, "we have two words for the things that we prove, proposition and theorem."[22] Now, "what we prove, in particular, the premises and conclusion of a logical inference"[22] are not called propositions, but judgments or assertions.There is one technicality here that one might not even notice. Strictly speaking, the word

judgement, or assertion, is used in particular for the premises and conclusion of a logical inference, where it usually means an affirmation or denial. Most of modern logic gets along with just affirmations. A formula is not affirmed directly, but it has to be *grasped* as a proposition and that proposition then can be affirmed. When

$$\frac{A\ \mathsf{prop} \qquad B\ \mathsf{prop} \qquad A\ \mathsf{true}}{A\ \vee\ B\ \mathsf{true}}$$

should be a *rule of disjunction introduction*, then grasping A and B as propositions, A prop and B prop do figure as premises for that rule although they are not an affirmation nor a denial. Martin-Löf extends a use of the word *judgment* to include such new *forms of judgment*, which are not only affirmations or denials anymore. Extending that usage allows us to denote premises and conclusion of an inference as *judgments of some specific form*. This wording is important because for typed λ-calculus our goal is the derivation of judgments the form $\Gamma \vdash T : a$, which means T has type a in context Γ. These judgments appear as the premises and conclusion of *type checking rules of inference*. There are three basic introduction type checking rules of typed λ-calculus: introducing λ-abstraction, introducing function application, and introducing variable usage.

3.2 | Typed λ-calculus

In order to formalize the previously motivated application in Section 2, we spoke about *univariate* and *multivariate* functions, *tuples* made of *scalars* and *indices* for various operations on tuples and multivariate functions. These all make valid *types* in our consideration and therefore we model a "Type" in our augmented λ-calculus to be introduced by the following introduction rules:

$$\frac{k : \mathbb{N}}{\mathsf{index}\ k :\ \mathsf{Type}}\ (\mathsf{index}\ -)$$

$$\frac{}{\mathsf{fun11}\ :\ \mathsf{Type}}\ (\ \mathsf{fun11}\ )$$

$$\frac{m : \mathbb{N}}{\mathsf{funM1}\ m :\ \mathsf{Type}}\ (\mathsf{funM1}\ -)$$

$$\frac{n : \mathbb{N}}{\mathsf{fun1N}\ n :\ \mathsf{Type}}\ (\mathsf{fun1N}\ -)$$

$$\frac{m : \mathbb{N} \qquad n : \mathbb{N}}{\mathsf{funMN}\ m\ n :\ \mathsf{Type}}\ (\mathsf{funMN}\ -\ -)$$

$$\frac{k : \mathbb{N}}{\mathsf{tuple}\ k :\ \mathsf{Type}}\ (\mathsf{tuple}\ -)$$

$$\frac{}{\mathsf{scalar}\ :\ \mathsf{Type}}\ (\ \mathsf{scalar}\ ).$$

These mean, that there *is* a type for univariate functions "fun11" and a type for scalars. For every natural number, there is one type of indices, one type of functions taking *m* arguments to a single output "funM1," and one type of functions taking a single input to *n* outputs "fun1N." For every two natural numbers *m* and *n*, there is a function type taking *m* inputs to *n* outputs "funMN." These are purely syntactical introduction rules that are named semantically but do not have their intended meaning yet. However, this "Type" serves as index set over which we will define *the family of valid terms* meaning we regard the totality of terms partitioned by their "Type."

These rules, in a very exact sence, correspond to a datatype definition in the Agda[10] language[29].

We have, that for every "Type" that can be introduced by our stated introduction rules, there is exactly one element in the datatype that we have defined within the Agda language and vice versa. This property makes it suitable to support our formalization as we go along. Here $\mathbb{N}$ is inductively defined in the usual way which is not much of interest here. For the formalization we introduced also the totality "Name" of names where variables are chosen from, but this also a minor point.

Some of the introduction type checking rules of λ-calculus in general are λ-abstraction and function application. Where the latter usually is denoted just by juxtaposition, without an explicit operator, we emphasize this by

the use of ⦇ and ⦈. The first rule of λ-abstraction for a and b being types in our λ-calculus and x being a name is written as.

$$\frac{\Gamma, (x, a) \vdash T : b}{\Gamma \vdash \lambda x \,.\, T : a \to b}\ (\lambda - \ . -).$$

It takes us from the judgment that "in a context consisting of first, Γ and second, the variable x being of type a, within that context the term T is of type b" to the judgment that "in context Γ the term $\lambda\ x\ .\ T$ is of type $a \to b$."

In our application in Section 2, we only needed to abstract over scalars or tuples, so a much stricter rule can be used for a formalization. We have chosen four much simpler rules instead which fix the types to the four combinations of tuples and scalars. You can find them in the Appendix C. The reason to take this simplification is that a following interpretation in section 4 will become easier having the types fixed. This is possible because we are not targeting to formalize a general purpose programming language, but rather a very specific one just targeting the partial derivative.A second rule introduces function application:

$$\frac{\Gamma \vdash T : a \to b \qquad \Gamma \vdash U : a}{\Gamma \vdash T ⦇\, U \,⦈ : b}\ (- \ ⦇ - ⦈)$$

It takes us from the two judgments that "*in a context Γ the term T is of type $a \to b$*" and "*in the same context Γ the term U is of type a*" to the judgment "*in context Γ again, $T ⦇\, U \,⦈$ is of type b.*" In our formalization we chose to have four such introduction rules operating on the corresponding argument types. One for each type of function.

3.3 | De Bruijn indices

There is one very basic key technique to work out for developing sane introduction rules that really respect the typing of variables with respect to some *context* Γ. It is noteworthy, that this technique is necessary to produce correctness guarantees from a programming language's type checker as motivated in Section 1.3. Unfortunately it is only expressable in a dependently typed programming language. In other programming languages, the following rules reduce to a list data structure.As introduction rules for a context we chose that there *is* an empty context

$$\frac{}{[] : \mathsf{Context}}\ ([])$$

and, when given a context Γ, we can form a new one $\Gamma, (x, a)$ for every name-type combination (x, a).

$$\frac{\Gamma : \mathsf{Context} \qquad x : \mathsf{Name} \qquad a : \mathsf{Type}}{\Gamma, (x, a) : \mathsf{Context}}\ (-, (-, -))$$

These rules make a context to *a list of tuples containing a name and a type* in our consideration. The de Bruijn indices[23] that we are going to work out will be indices that are guaranteed by their type to really point to a specific name-type combination within such context. One could even model a context as a list of just types and without the names. Within the Agda formalization we found it very expressive to have this redundant piece of information available. Still, a variable is to be identified by its de Bruijn index and not by its name.The first rule introduces a judgment that "de Bruijn index zero is an element of the type of de Bruijn indices that show the first name-type combination of a context being in that context."

$$\frac{\Gamma : \mathsf{Context} \qquad x : \mathsf{Name} \qquad a : \mathsf{Type}}{\mathsf{zero} : (x, a) \in \Gamma, (x, a)}\ (\mathsf{zero})$$

The second rule introduces a judgment that "an incremented de Bruijn index shows that a name-type combination is part of an appended context, given that it did so before."

$$\frac{i^b : (x, a) \in \Gamma \qquad y : \mathsf{Name} \qquad b : \mathsf{Type}}{\mathsf{suc}\ i^b : (x, a) \in \Gamma, (y, b)}\ (\mathsf{suc}-)$$

With these rules it is possible to give meaning to the last standard introduction type check rule of λ-calculus. It introduces the judgment that "in context Γ the variable x is of type a because of $\because$ the de Bruijn index i^b."

$$\frac{i^b : (x, a) \in \Gamma}{\Gamma \vdash x\ \because\ i^b : a}\ (- \because -)$$

Where this matches closely our Agda formalization, the rule is sometimes written more intuitively as.

$$\frac{x : a \in \Gamma}{\Gamma \vdash x : a}\ (\mathsf{Var})$$

or even.

$$\frac{x \in \Gamma}{\Gamma \vdash x : \Gamma(x)}\ (\mathsf{Var}).$$

3.4 | Augmenting λ-calculus

By just *using* the λ-calculus we got into the previous three rules and the use of de Bruijn indices even without any specifics from our application. After paying that entry fee for which do not exist many alternatives, we can finally work-in our application specific operations, which are substitution of the ith component, projecting out the ith component, composition of functions, the univariate derivative, and scalar multiplication.

$$\frac{\Gamma \vdash T : a \qquad i : \mathbb{N} \qquad \Gamma \vdash U : b}{\Gamma \vdash T[\bullet i := U] : a}\ (-[\bullet - := -])$$

$$\frac{\Gamma \vdash T : a \qquad i : \mathbb{N}}{\Gamma \vdash T \hat{} i : b}\ (- \hat{} -)$$

$$\frac{\Gamma \vdash T : b \to c \qquad \Gamma \vdash U : a \to b}{\Gamma \vdash T \circ U : a \to c}\ (- \circ -)$$

$$\frac{\Gamma \vdash T : r \to r}{\Gamma \vdash T' : r \to r}\ (-')$$

$$\frac{\Gamma \vdash T : r \qquad \Gamma \vdash U : r}{\Gamma \vdash T \cdot U : r}\ (- \cdot -)$$

For the previously introduced types in our specific λ-calculus, we introduced six rules for substitution, six rules for projection, four obvious rules for composition of functions, and one rule for the univariate derivative as well as one rule for scalar multiplication. You can find the rules in Appendix C and the Agda datatype of all well-formed λ-calculus terms is in Figure 7.

```
data Type : Set where
  index  : ℕ → Type
  fun11  : Type
  funM1  : ℕ → Type
  fun1N  : ℕ → Type
  funMN  : ℕ → ℕ → Type
  tuple  : ℕ → Type
  scalar : Type
```

FIGURE 6 Agda datatype of custom types to be regarded in an augmented λ-calculus

```
data Term (Γ : Context) : Type → Set where
  -- variables (come with their de Bruijn index i)
  _∴_ : ∀{a} → (x : Name) → (i^b : (x , a) ∈ Γ) → Term Γ a
  -- quantifiers over tuples and scalars
  λ^t_^t_._ : ∀{n} → (m : ℕ) → (x : Name) → Term ((x , tuple m) :: Γ) (tuple n) → Term Γ (funMN m n)
  λ^t_^s_._ :        (m : ℕ) → (x : Name) → Term ((x , tuple m) :: Γ) scalar    → Term Γ (funM1 m)
  λ^st_._   : ∀{n}           → (x : Name) → Term ((x , scalar ) :: Γ) (tuple n) → Term Γ (fun1N n)
  λ^ss_._   :                  (x : Name) → Term ((x , scalar ) :: Γ) scalar    → Term Γ fun11
  Σ_^s_._   :        (k : ℕ) → (x : Name) → Term ((x , index k) :: Γ) scalar    → Term Γ scalar
  -- application
  _^11(_) :          Term Γ fun11       → Term Γ scalar    → Term Γ scalar
  _^m1(_) : ∀{m}   → Term Γ (funM1 m)   → Term Γ (tuple m) → Term Γ scalar
  _^1n(_) : ∀{n}   → Term Γ (fun1N n)   → Term Γ scalar    → Term Γ (tuple n)
  _^mn(_) : ∀{m n} → Term Γ (funMN m n) → Term Γ (tuple m) → Term Γ (tuple n)
  -- substitution
  _^k[•_:=_]  : ∀{k}   → Term Γ (tuple k)   → Fin k           → Term Γ scalar → Term Γ (tuple k)
  _^n[•_:=_]  : ∀{n}   → Term Γ (fun1N n)   → Fin n           → Term Γ scalar → Term Γ (fun1N n)
  _*[•_:=_]   : ∀{m n} → Term Γ (funMN m n) → Fin n           → Term Γ scalar → Term Γ (funMN m n)
  _^ki[•_:=_] : ∀{k}   → Term Γ (tuple k)   → Term Γ (index k) → Term Γ scalar → Term Γ (tuple k)
  _^ni[•_:=_] : ∀{n}   → Term Γ (fun1N n)   → Term Γ (index n) → Term Γ scalar → Term Γ (fun1N n)
  _*^i[•_:=_] : ∀{m n} → Term Γ (funMN m n) → Term Γ (index n) → Term Γ scalar → Term Γ (funMN m n)
  -- projection
  _^^k_  : ∀{k}   → Term Γ (tuple k)   → Fin k           → Term Γ scalar
  _^^n_  : ∀{n}   → Term Γ (fun1N n)   → Fin n           → Term Γ fun11
  _^^*_  : ∀{m n} → Term Γ (funMN m n) → Fin n           → Term Γ (funM1 m)
  _^^ki_ : ∀{k}   → Term Γ (tuple k)   → Term Γ (index k) → Term Γ scalar
  _^^ni_ : ∀{n}   → Term Γ (fun1N n)   → Term Γ (index n) → Term Γ fun11
  _^^*i_ : ∀{m n} → Term Γ (funMN m n) → Term Γ (index n) → Term Γ (funM1 m)
  -- composition
  _○^111_ :            Term Γ fun11       → Term Γ fun11       → Term Γ fun11
  _○^1k1_ : ∀{k}     → Term Γ (funM1 k)   → Term Γ (fun1N   k) → Term Γ fun11
  _○^mkn_ : ∀{m k n} → Term Γ (funMN k n) → Term Γ (funMN m k) → Term Γ (funMN m n)
  _○^m1n_ : ∀{m n}   → Term Γ (fun1N   n) → Term Γ (funM1 m)   → Term Γ (funMN m n)
  -- special functions
  _'^s  : Term Γ fun11  → Term Γ fun11
  _·^s_ : Term Γ scalar → Term Γ scalar → Term Γ scalar
```

FIGURE 7 The datatype "Term" of well-formed terms for an *augmented* λ-calculus suitable to express the partial derivative within the Agda programming language. Fin k is the type of natural numbers less than k, sometimes denoted $\mathbb{N}_k$ or $\mathbb{N}_{<k}$

3.5 | Chain of justification

All the data structures and data transformations described in Section 2 represent computations for the partial derivative function. However, even after translating them into an augmented λ-calculus, they are not yet more than the mere

skeletons carrying around metadata. All the transformations we now implement on these λ-terms, which should respect this, yet hypothetical, computation are just operations transforming that metadata.

The resulting λ-terms can only be turned into a computation when a lower layer, that is, an implementation providing the univariate derivative and a representation of functions providing these computations, is present such that the terms can be interpreted, that is, turned into a computation and executed.

There are just a few properties even possible to be proven without further assumption at this high level. We have made the distinction between a computational equivalence $\equiv$ that is justified within our investigation by the computational equivalences of the λ-calculus and the propositional equality $=$ that is used when a property of the univariate derivative $'$, that operation we presupposed for our whole consideration, was made use of.

For the computational equivalences $\equiv$, there is some chance to express those in terms of α-conversion, β-reduction, and η-reduction. However, the provability of the equivalences denoted by $=$ depends on the underlying *interpretation*.

Consistency of computational equivalence resulting from the presented transformations depends on a consistent implementation of the considered layer, of course, and precisely on a consistent implementation of these two equality-transformations of the lower layer. These two equality-transformations are in some sence *dependencies* of our considered layer. The benefit is that the implementation of the considered layer can be verified in a way independently from a lower level application increasing the overall trust and decomposing monolithic software ventures into more modular ones. Similar to the two assumptions $=_{\text{lin}}$ and $=_{\text{chain}}$ of the univariate derivative, it is possible to determine additional assumptions that are necessary in proofs of additional theorems. In Section 4, we give guidance how these rather abstract assumptions become more concrete with a chosen interpretation for the augmented λ-calculus.

4 | SOFTWARE ARCHITECTURE

In Section 3, a family of datatypes for well-formed terms in an augmented λ-calculus was set up. That family was indexed by a "Type" being the term's type and a "Context," to realize a valid use of variables. This should serve as a foundation for an implementation of our application from Section 2. Objects and equivalences from that application, the partial derivative and the multivariate chain rule, can be expressed within this λ-calculus. However, while guaranteeing these translations to be well-formed terms, this still does not make a computation. In Section 1.2, it was motivated how the formulation of the construction of a shape function element within a programming language easily leads to a metaimplementation. The metaimplementation's purpose is to generate an efficient implementation, where the metaimplementation itself should be focused on validity rather than efficiency. Our approach in Section 3 should enable to achieve a high validity in a metaimplementation.

We are now focusing on how to give the λ-terms a suitable interpretation: Our approach has changed the task of giving a direct interpretation to partial derivatives, into the task of giving a direct interpretation to some lower level primitives. These are variable access, quantification, substitution, projection, composition, and the two *special functions*, which were univariate derivative and a scalar multiplication. Replacing one notion of partial derivative[30], with these lot of operations seems quite a lot of machinery and not worth the trade.

At this point we argue that, first, this approach is in some sense minimal. Decomposing partial derivative into more basic notions as in Equation (23) involves only a notion of univariate derivative of a certain function[31]. With our elaboration in Section 2, we collected what obligations arise when working out such decomposition in a way, precise enough to reach a level of "correctness and completeness necessary to get a computer program to work."[21] This level might be "a couple of orders of magnitude higher"[21] than the level it needs to convince humans.

Second, one might *have* nothing more than coordinate transition functions in an implementation on a machine as representations of the *data* of boundary value problems. Recall that motivation from Section 1.1 which promised one generic rule for coordinate transformations, indexed by three indices (p, q, ω). These should cover the electromagnetic quantities of interest that will show up in an implementation, applying techniques mentioned in Section 1.2. There we have identified as an important ingredient the *degrees of freedom* which were mappings from a polynomial differential form u to a real number:

$$u \mapsto \int_f \left(\operatorname{tr}_f u\right) \wedge q : \mathcal{P}_\bullet \Lambda^k(T) \to \mathbb{R}.$$

Here we would identify u as the section of an associated bundle as in Section 1.1. Since this section might not be directly representable within the machine's memory, we would handle a coordinate representation of it instead. This was denoted $\overset{xyz}{\sigma}$ in Figure 2 or phys. field numbers correspond to xyz in Figure 8.

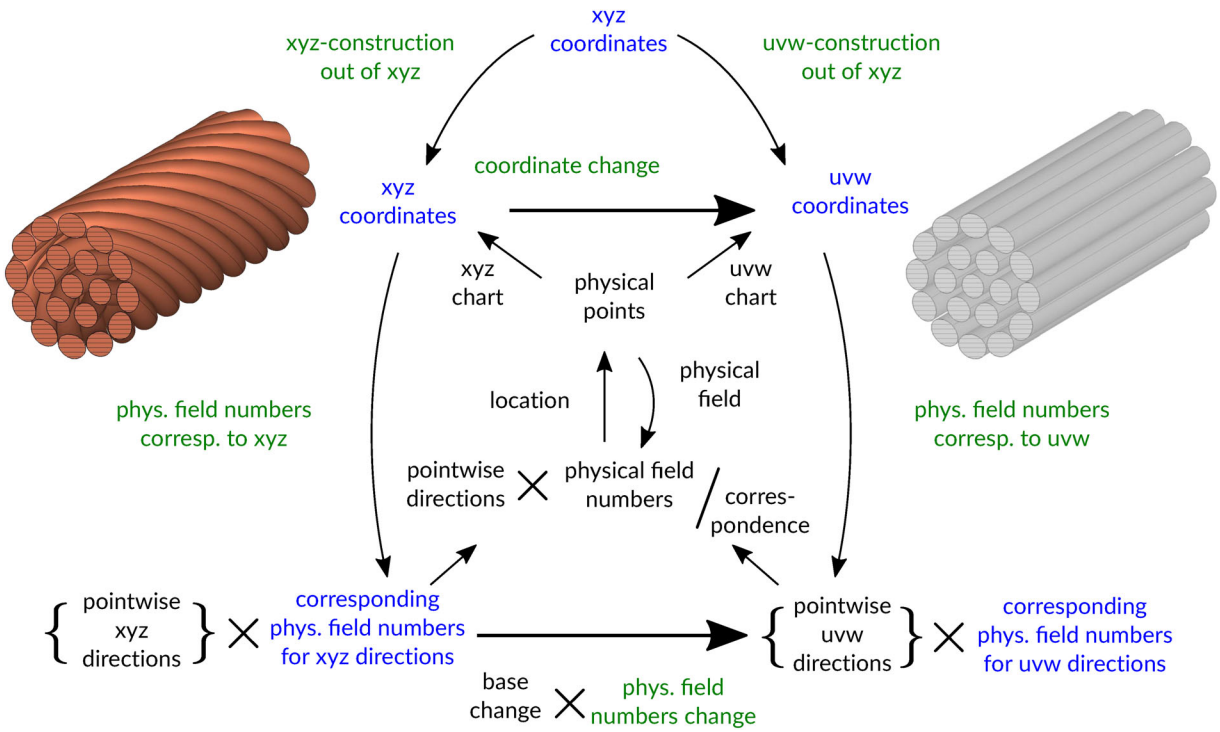

FIGURE 8 Objects in a computer program involved in an electromagnetic transformation. The blue bits tag number data used when requesting particular physical field numbers for example, for computing a numerical quadrature. The green bits tag predominant computations and fields necessary to be represented in the computer's memory. It is important to note, that an electromagnetic field quantity is represented as one of the green bits, even though its contained degrees of freedom are thought of being blue bits at first

Evaluation of these integrals in order to produce data for a discrete linear system to be solved is usually done by numerical quadrature. That quadrature *queries* the physical field quantitie's value at some specific coordinates. If one cannot, or does not want to, predict which physical field values will be queried, it might appeal to represent the physical field as a computation. Of course, that computation might internally interpolate field values at some specific coordinates with polynomials, as is the case with the polynomial differential forms.

When speaking of finite elements, these are usually transformed to a *reference element* already in order to evaluate a numerical quadrature. Such approach is desired, since it allows the quadrature to be implemented in a fixed way with precalculated coefficients. Therefore, we might say that every computer program implementing this technique has to deal with partial derivatives in some way already. However, having just one integral transformation might not be worth the effort we have made in the previous part of this article. We think that a construction of the boundary value problem itself might be given in terms of a longer chain of composed coordinate transformations. The representation of a boundary value problem could then be internally encoded as an equivalence transformation out of primitives. However, this is just a motivation for our approach.

We argue it to be, at least, a justified perspective that these kinds of coordinate transformations[32] apply for a wide range of *numerical software* as sketched in Section 1.2. Here the particular focus was on boundary value problems expressable by some Hodge-Laplacian over a manifold, discretized with a simplicial complex.

In a broader sense, we understand a part of *software* as a mapping of mathematical models for coordinate transformations, given by (1) in terms of the partial derivative, into a programming language. This is done by decomposing this mapping into first, a mapping of partial derivative into a formal language based on λ-calculus and second, a mapping of these λ-calculus terms to computations in a concrete programming language. Benefit arises, since the first mapping can be discussed and justified on a theoretical basis, where the second is much more arbitrary in its nature: arbitrary in a sense that we have to deal with different computational models to create programs that run on different kinds of machines. Now, we will elaborate on some of these more arbitrary ways to map our specific λ-terms to a concrete programming language, or rather map them to a model of evaluation coming with such concrete programming language. A concrete programming language for that purpose comes with a syntax *and* an evaluation strategy.

4.1 | Interpreting λ-terms formally

A straight-forward way, since we already have an Agda formalization, would be to continue here by implementing an evaluation function. Doing so is a typical task and two new concepts occur: for a context Γ and one of our custom types a, the evaluation function "eval" maps an *environment* of that context and a λ-term of that type and context to a *value* of the *interpretation* of a.

$$\text{eval} : \text{Environment}\ \Gamma \to \text{Term}\ \Gamma\ a \to \text{Interpretation}\ a$$

Here, "Interpretation" maps our custom types from Figure 6 to types of the Agda language, which are elements of the universe "Set":

$$\text{Interpretation} : \text{Type} \to \text{Set}.$$

An interpretation of our λ-term's types then really is captured by a function from our previously defined "Type" to the types of the programming language, which is Agda in this case.

We showed in Section 3 that a context—in the way we introduced it—can be regarded as a list holding multiple variable name and type combinations. An *environment* for such context can be regarded as holding the corresponding values of these types. We might only use an environment by *looking up variables*, which are de Bruijn indices in our formalization:

$$\text{lookup} : \text{Environment}\ \Gamma \to a \in \Gamma \to \text{Interpretation}\ a.$$

This shows that the choice of implementing an environment is already a little less *fixed*. We could mimick the context and use a list, but in contrast to the context, the environment will be present in our evaluation's *computation* where we might forget about the context completely. One might prove within Agda that an implementation of lookup never fails when given a valid de Bruijn index and then strip all the type information, revealing bare computations. Therefore with the environment, we do want to incorporate some aspects of performance. Speaking of performance, we might have a hard time continuing to use the Agda language itself as a target for evaluation. It is possible to implement a numerical software within this language[33], but this programming language's environment offers only a limited help to put the machine into its most efficient state for the purpose of a computation. A typical goal is to map scalars to *unboxed*[34] floating point machine numbers, which lack a lot of the properties of their counterparts from $\mathbb{R}$. The Agda programming language offers more support for showing properties on rationals or constructive real numbers. Unfortunately, these numbers tend to have a representation making them unsuited for numerical computations. However, that is a usual trade we have a lot with numerical algorithms: once their theory is worked out for exact real numbers and the algorithm is *stable*, then we do apply it on inexact floating-point machine numbers, fingers crossed[35].

What we also get with the evaluation as a mapping is the possibility to proof preservation of the $=_{\text{lin}}$ and $=_{\text{chain}}$ equivalences from Section 2 for our intended implementation. If we chose to interpret our custom λ-calculus functions really as functions, then in particular the univariate derivative might be chosen to be an inexact black-box operation such as the difference quotient. In that case, the chances are high that we will lose the possibility to exactly proof that $=_{\text{lin}}$ and $=_{\text{chain}}$ are preserved by our evaluation function even if operating on exact rational numbers. That could be intended and we might formally track error bounds with all our operations to proof that the error resulting from this operation amortizes comparing to some other error. However, the more interesting case would be to interpret λ-functions not as functions but as data structures with a more *interesting* interpretation of λ-function-application as a data transformation. This perspective is elaborated in Section 4.3.

4.2 | Interpreting λ-terms less formally

One might have a formal model of a programming language at hand such that λ-terms can be translated. Depending on the degree of formalism, surjectivity of the eval function can be proven or even that the resulting terms are still well-defined in the target language. This is essentially some type of code generation, where the weakest variant would be to interpret all our custom λ-terms within the string monoid, calling it "*code*." Even if one does not formally model the target language, this still gives a possibility to implement transformations at the λ-term level, before they are evaluated to

code. Although introducing a large margin for interpretation and bugs, this approach could be well suited for generating code running in a very limited, for example, lock-step, environment.

4.3 | Interpreting λ-terms as data transformations

As motivated before, an interpretation of λ-functions to data structures of a target language might currently be the most rewarding one. For the finite element spaces from our application, a lot of different multivariate polynomial functions have to be operated with. These functions can be represented by polynomial coefficients, together with a custom function application operation that does use these coefficients to compute the polynomial. Furthermore, the univariate derivative operation on such coefficient representation is not only a very cheap one but also exact when using exact number representations. That enables to proof the eval function to preserve $=_{\text{lin}}$ and $=_{\text{chain}}$ computationally.

In a metaimplementation for generating an efficient implementation it seems reasonable to start out with an initial candidate for the implementation and then apply rewrite rules to optimize this implementation. Implementing correct rewrites and data transformations is where Agda, and functional programming in general, shines. The reason for that is an inductive definition of the data structures in question which enables an *exhaustion check* to proof functions to be total, that is, not having missed a case. This usually pays off in case-analysis-heavy applications such as designing domain specific languages, as we do here, or improving the encoding of a data structure. As for multivariate polynomials, these can obviously be represented as *packed* chunks of computer memory, holding their coefficients. However, we might add some information or invent an interesting reference type for better tying them to the simplicial complex they are originating from. Having equivalence proven for one *obvious* encoding it can be easier for a new encoding to show it isomorphic, transferring the proofs.

5 | CONCLUSION

We have explained transformations on the partial derivative in terms of computational notions from λ-calculus with an additional term substitution. This mechanism has been implemented to generate listings for the general case as in Appendix A and for all concrete multivariate cases, indexed by $\mathbf{j} \in \mathbb{N}$ and $\mathbf{i} \in [1,\mathbf{j}]$, exemplary for $\mathbf{j} = 2$ and $\mathbf{i} = 1$ as in Appendix B, out of the same internal representation. It was argued, what general obligations arise when translating the theory into a computational layer of abstraction, for which the λ-calculus served as a model. We showed how a translation into an augmented λ-calculus can be formalized within type theoretical terms and implemented that formalization in the Agda programming language. Finally, we gave some examples how to make use of the presented approach and favorized one particular possibility. Our current research is about this exact undertaking and the foundational considerations are shown in our contribution.

Small programs as well as big software, no matter whether directly implementing this layer or not, will suffer from the inevitable tediousness of coordinate transformations when exploiting these techniques too much. That does not pose a problem when being aware of this issue and actively increasing rigor if this kind of complexity gets out of control. We have presented a way to establish that direction of rigor, motivated by the application of encoding the transformation laws common to the electromagnetic theory. Accompanying that way is an interpretation to guide an implementation demanding it.

ORCID

Marcus Christian Lehmann https://orcid.org/0000-0002-5881-1709
Mirsad Hadžiefendić https://orcid.org/0000-0003-3304-2306

ENDNOTES

[1]See Section 1.1

[2]p 21, Hehl[7].

[3]p 27, Hehl [7].

[4]Adapted from, p 212-214, Baez [8].

[5]References are usually represented as integers indicating an item over some universe, most commonly an enumeration of cases or an address of the machine's memory.

[6]The spaces V_h do not necessarily have to be subspaces of V.

[7]The buildup of a matrix for the resulting discrete linear system is also called assembly process.

[8]*Finite element exterior calculus* NSF/CBMS course, ICERM, 2012; lecture 9, minutes 40 seconds 10.

[9]A limited amount is shown in a table from the original article[12] and the various families of bases are implemented within the FENICS[24] project. Reproducing the bases from their article required us some amount of bookkeeping.

[10]In our reference[16], this is in preparation for explaining Curry-style and Church-style λ-calculi.

[11]Encoding of de Bruijn indices[23] can be regarded with respect to that interpretation, but does not need to. A further treatment of defining equivalences and proving their preservation, makes the *encoding* of propositions as types indispensable. Therefore, propositions become common objects to handle in a formal implementation.

[12]A.1.9 A notation for functions

[13]Bossavit uses a straight-arrow $\to$ for both, function abstractions and function types, whereas we would argue to use the barred-arrow $\mapsto$ for function abstractions and $\to$ for function types.

[14]In programming languages, this is sometimes called a *wrapper function*.

[15]We use the nomer multivariate, although it usually denotes functions taking multiple arguments to one result. Since in our case, the results are not correlated to each other, and functions that give multiple uncorrelated results can be represented as a collection of these multivariate functions in the usual sense, we do not distinguish the terms here that much.

[16]Our proposed variant is mostly borrowed from the *parameter-pack expansion* which is a carefully specified notation that appeared in the standard of the C++ programming language[13] first in its 2011 version. A parameter-pack can only appear in a metacomputation expressed within the templating system of C++. This notation is implemented in all current compilers complying to that standard.

[17]Just like a quantifier the tuple expansion binds unbound tuples where the unbound tuples are underlined.

[18]This use is also borrowed from the C++ programming language standard.

[19]This means especially that we do not make use of *currying* to express the multivariate functions. Furthermore as it turns out, the necessary (tuple-) arity of functions within this article is always one.

[20]Which is usually the case in programming languages tagged *imperative*

[21]Which is usually the case in programming languages tagged *functional*: a function takes exactly one argument, which might be a tuple, and there is no difference of *something* and the one-tuple of *something*

[22]In theoretical computer science this is usually realized not by shadowing, but by limiting the α-equivalence to the cases where the argument x of $x \mapsto \mathbf{term}$ does not occur as a free variable of the **term**, which is stated as $x \notin \mathrm{FV}(\mathbf{term})$. However, shadowing exists in the most programming languages.

[23]One distinguishes the function-level partial derivative ∂_i with respect to the ith argument of a function from the vector field $\partial/\partial x^i$ induced by the ith coordinate x^i, where both fulfill the rules of what it means to be called a *derivative*.

[24]This is just the other way around as for the *basis*, where J_f transforms the A-induced basis into the B-induced basis.

[25]That is, a style where no arguments $(\underline{x}...)$ are present

[26]The direction, that is, *from the A coordinate system to the B coordinate system* or *in the direction that f is defined*, is meant here. To emphasize its distinction from the physical direction in space, we call it the *logical* direction instead.

[27]Martin-Löf attributes it to Russell, translating Frege's *Urteil* into *assertion*, and calling the combination of Frege's judgement stroke "|" and content stroke "-" the assertion sign "$\vdash$."

[28]That use of the word *proposition* is again attributed to Russell

[29]The Agda language, on the one hand can be introduced as a functional programming language that, on the other hand, is powerful enough to express constructive mathematics. Agda builds on top of a type theory, as introduced by Martin-Löf. It supports dependently typed pattern matching, using so-called *Miller pattern unification*, with Σ-types, inductive datatypes and universe polymorphism.

[30]Or an evaluation of Jacobian matrices if you like so

[31]Which is correspondence with what Spivak mentioned as *ordinary* derivative and cited in Equation (8)

[32][or integral transformations if you like so]

[33]The Agda language is implemented in Haskell and can use Haskell methods, which in turn via a *foreign function interface* can call arbitrary system libraries

[34]Programming languages with automatic reference and memory management tend to implicitly attach typing information to a value to be able treating this value via references instead. This is done because references into memory on a machine have a uniform representation. It is called "boxing" of a value. Boxing often demands memory allocation. Preventing frequent memory allocation, for example, per-scalar memory allocation, is very important to achieve high efficiency in an implementation.

[35]Unless using promising techniques such as interval arithmetic to provide guarantees for this approach

[Correction added on 10 July 2020, after first online publication: ENDNOTES number 32 has been corrected to '[or integral transformations if you like so]' in this version.]

REFERENCES

1. Bossavit A. On the notion of anisotropy of constitutive laws: some implications of the 'Hodge implies metric' result. *COMPEL - Int J Comput Math Electr Electron Eng*. 2001;20(1):233-239.
2. Dantzig D. The fundamental equations of electromagnetism, independent of metrical geometry. *Math Proc Camb Philos Soc*. 1934;30(4): 421-427.
3. Bossavit A. The premetric approach to electromagnetism in the 'waves are not vectors' debate. *Adv Electromagnetics*. 2012;1(1):97-102.
4. Raumonen P. *Mathematical Structures for Dimensional Reduction and Equivalence Classification of Electromagnetic Boundary Value Problems*. Tampere, Finland: Tampere University of Technology; 2009.
5. Spivak Michael. *Calculus on Manifolds*. W. A. Benjamin; New York, NY; 1965.
6. Tonti E. *The Mathematical Structure of Classical and Relativistic Physics*. Basel, Switzerland: Birkhäuser; 2013.
7. Hehl FW, Obukhov YN. *Foundations of Classical Electrodynamics*. Boston, MA: Birkhauser; 2003.
8. Baez J, Muniain JP. *Gauge Fields, Knots and Gravity*. Singapore: World Scientific Publishing Company; 1994.
9. Auchmann Bernhard, Kurz Stefan. Observers and splitting structures in relativistic electrodynamics. *J Phys A Math Theoret* 2014;47: 435202.
10. Norell U. Dependently typed programming in Agda. In: Koopman P, Plasmeijer R, Swierstra D, eds. *Advanced Functional Programming. AFP 2008*. Lecture Notes in Computer Science, Vol 5832. Berlin, Heidelberg: Springer; 2009.
11. Arnold Douglas N., Falk Richard S., Winther Ragnar Finite element exterior calculus: from Hodge theory to numerical stability. *Bull Amer Math Soc* 2010;47:281–354.
12. Arnold DN, Falk RS, Winther R. Geometric decompositions and local bases for spaces of finite element differential forms. *Comput. Methods Appl Mech Engrg*. 2009;198:1660-1672.
13. ISO. *ISO/IEC 14882:2017 Information technology—Programming languages—C++*. fifth ed. Geneva, Switzerland; 2017.
14. Pellikka M, Tarhasaari T, Suuriniemi S, Kettunen L. A programming interface to the Riemannian manifold in a finite element environment. *J Comput Appl Math*. 2013;246:225-233.
15. Church A. An unsolvable problem of elementary number theory. *American Journal of Mathematics*. 1936;58(2):345-363.
16. Barendregt P. Henk *The lambda calculus: its syntax and semantics*. North-Holland Publishing Company; sole distributors for the USA and Canada Elsevier North-Holland Amsterdam; Amsterdam: North-Holland; 1981.
17. Wadler P. Propositions as types. *Commun ACM*. 2015;58(12):75-84.
18. Bossavit A. Appendix A— mathematical background. In: Bossavit A, ed. *Computational Electromagnetism*. San Diego, CA: Academic Press; 1998:263-318.
19. Wolfram Research Inc. Mathematica 11.0 2018.
20. Martin-Löf P. *Constructive Mathematics and Computer Programming*. Upper Saddle River, NJ: Prentice-Hall, Inc.; 1985:167-184.
21. Thurston WP. On proof and progress in mathematics. *Bull Amer Math Soc (NS)*. 1994;30:161-177.
22. Martin-Löf Per. On the meanings of the logical constant and the justifications of the logical laws. Technical Report 2. Scuola di Specializziazione in Logica Matematica, Università di Siena.
23. Bruijn NG. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*. 1972;75(5):381-392.
24. Alnæs MS, Blechta J, Hake J, et al. The FEniCS project version 1.5. *Arch Numer Softw*. 2015;3(100):9-23.

How to cite this article: Lehmann MC, Hadžiefendić M, Piwonski A, Schuhmann R. Encoding electromagnetic transformation laws for dimensional reduction. *Int J Numer Model*. 2020;33:e2747. https://doi.org/10.1002/jnm.2747

APPENDIX

In the appendix, we give a listing of the computational equivalences used to demonstrate the dependencies of the notion of partial derivative and the chain rule of the partial derivative on the notion of univariate derivative and the corresponding univariate chain rule. Both listings have been created out of the same internal representation with the rules of parameter-pack expansion borrowed from the C++ programming language, with the help of our own implementation of the parameter-pack expansion, supporting the mentioned substitution. For the expanded listing in Appendix B, we chose $f, g : \mathbb{R}^2 \to \mathbb{R}^2$ and $\mathbf{i} = 1$.

Furthermore, we attached a translation of the Agda datatype of well-formed λ-terms from Figure 7.

APPENDIX A

$$\frac{\partial\left(f^{\mathbf{j}} \circ g\right)}{\partial x^{\mathbf{i}}} \tag{A1}$$

$$\equiv_{\text{def}} \quad (\underline{x}\ldots) \mapsto \left(\mathbf{z} \mapsto \left(f^{\mathbf{j}} \circ g\right)\left(\underline{x}\ldots\left[\bullet^{\mathbf{i}} := \mathbf{z}\right]\right)\right)'\left(x^{\mathbf{i}}\right) \tag{A2}$$

$$\equiv_{\circ} \quad (\underline{x}\ldots) \mapsto \left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(\underline{g}\left(\underline{x}\ldots\left[\bullet^{\mathbf{i}} := \mathbf{z}\right]\right)\ldots\right)\right)'\left(x^{\mathbf{i}}\right) \tag{A3}$$

$$=_{\text{lin}'} \quad (\underline{x}\ldots) \mapsto \sum_k \left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(\underline{g}(\underline{x}\ldots)\ldots\left[\bullet^{k} := g^{k}\left(\underline{x}\ldots\left[\bullet^{\mathbf{i}} := \mathbf{z}\right]\right)\right]\right)\right)'\left(x^{\mathbf{i}}\right) \tag{A4}$$

$$\equiv_{\circ} \quad (\underline{x}\ldots) \mapsto \sum_k \left(\overbrace{\left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(\underline{g}(\underline{x}\ldots)\ldots\left[\bullet^{k} := \mathbf{z}\right]\right)\right)}^{a^k} \circ \overbrace{\left(\mathbf{z} \mapsto g^{k}\left(\underline{x}\ldots\left[\bullet^{\mathbf{i}} := \mathbf{z}\right]\right)\right)}^{b^k}\right)'\left(x^{\mathbf{i}}\right) \tag{A5}$$

$$\equiv_{\text{def}} \quad (\underline{x}\ldots) \mapsto \sum_k \left(a^k \circ b^k\right)'\left(x^{\mathbf{i}}\right) \tag{A6}$$

$$=_{\text{chain}'} \quad (\underline{x}\ldots) \mapsto \sum_k \left(\left(a^{k\prime} \circ b^k\right)\left(x^{\mathbf{i}}\right)\right) \cdot b^{k\prime}\left(x^{\mathbf{i}}\right) \tag{A7}$$

$$\equiv_{\text{def}} \quad (\underline{x}\ldots) \mapsto \sum_k \underbrace{\left(\left(\overbrace{\left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(\underline{g}\left(\underline{x}\ldots\right)\ldots\left[\bullet^{k} := \mathbf{z}\right]\right)\right)'}^{a^{k\prime}} \circ \overbrace{\left(\mathbf{z} \mapsto g^{k}\left(\underline{x}\ldots\left[\bullet^{\mathbf{i}} := \mathbf{z}\right]\right)\right)}^{b^k}\right)\left(x^{\mathbf{i}}\right)\right)}_{\frac{\partial f^{\mathbf{j}}}{\partial y^k}\left(g^k(\underline{x}\ldots)\right)} \cdot \underbrace{\overbrace{\left(\mathbf{z} \mapsto g^{k}\left(\underline{x}\ldots\left[\bullet^{\mathbf{i}} := \mathbf{z}\right]\right)\right)'\left(x^{\mathbf{i}}\right)}^{b^{k\prime}\left(x^{\mathbf{i}}\right)}}_{\frac{\partial g^k}{\partial x^{\mathbf{i}}}(\underline{x}\ldots)} \tag{A8}$$

$$\equiv_{\beta} \quad (\underline{x}\ldots) \mapsto \sum_k \left(\left(\left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(\underline{g}(\underline{x}\ldots)\ldots\left[\bullet^{k} := \mathbf{z}\right]\right)\right)' \circ \left(\mathbf{z} \mapsto g^{k}\left(\underline{x}\ldots\left[\bullet^{\mathbf{i}} := \mathbf{z}\right]\right)\right)\right)\left(x^{\mathbf{i}}\right)\right) \cdot \underbrace{\left(\left(\underline{y}\ldots\right) \mapsto \left(\mathbf{z} \mapsto g^{k}\left(\underline{y}\ldots\left[\bullet^{\mathbf{i}} := \mathbf{z}\right]\right)\right)'\left(y^{\mathbf{i}}\right)\right)}_{\frac{\partial g^k}{\partial x^{\mathbf{i}}}}(\underline{x}\ldots) \tag{A9}$$

$$\equiv_{\text{def}} \quad (\underline{x}\ldots) \mapsto \sum_k \underbrace{\left(\left(\left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(\underline{g}\left(\underline{x}\ldots\right)\ldots\left[\bullet^{k} := \mathbf{z}\right]\right)\right)' \circ \left(\mathbf{z} \mapsto g^{k}\left(\underline{x}\ldots\left[\bullet^{\mathbf{i}} := \mathbf{z}\right]\right)\right)\right)\left(x^{\mathbf{i}}\right)\right)}_{\frac{\partial f^{\mathbf{j}}}{\partial y^k}\left(g^k(\underline{x}\ldots)\right)} \cdot \frac{\partial g^k}{\partial x^{\mathbf{i}}}\left(\underline{x}\ldots\right) \tag{A10}$$

$$\equiv_{\circ} \quad (\underline{x}\ldots) \mapsto \sum_k \underbrace{\left(\left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(\underline{g}\left(\underline{x}\ldots\right)\ldots\left[\bullet^{k} := \mathbf{z}\right]\right)\right)'\left(g^k\left(\underline{x}\ldots\right)\right)\right)}_{\frac{\partial f^{\mathbf{j}}}{\partial y^k}\left(g^k(\underline{x}\ldots)\right)} \cdot \frac{\partial g^k}{\partial x^{\mathbf{i}}}\left(\underline{x}\ldots\right) \tag{A11}$$

$$\equiv_{\beta} \quad (\underline{x}\ldots) \mapsto \sum_k \underbrace{\left(\left(\underline{y}\ldots\right) \mapsto \left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(\underline{y}\ldots\left[\bullet^k := \mathbf{z}\right]\right)\right)'\left(y^k\right)\right)}_{\frac{\partial f^{\mathbf{j}}}{\partial y^k}}\left(\underline{g}(\underline{x}\ldots)\ldots\right) \cdot \frac{\partial g^k}{\partial x^{\mathbf{i}}}(\underline{x}\ldots) \tag{A12}$$

$$\equiv_{\text{def}} \quad (\underline{x}\ldots) \mapsto \sum_k \frac{\partial f^{\mathbf{j}}}{\partial y^k}\left(\underline{g}(\underline{x}\ldots)\ldots\right) \cdot \frac{\partial g^k}{\partial x^{\mathbf{i}}}(\underline{x}\ldots) \tag{A13}$$

$$\equiv_{\circ} \quad (\underline{x}\ldots) \mapsto \sum_k \left(\frac{\partial f^{\mathbf{j}}}{\partial y^k} \circ g\right)(\underline{x}\ldots) \cdot \frac{\partial g^k}{\partial x^{\mathbf{i}}}(\underline{x}\ldots) \tag{A14}$$

$$\equiv_{\text{def}} \quad (\underline{x}\ldots) \mapsto \sum_k \left(\left(\frac{\partial f^{\mathbf{j}}}{\partial y^k} \circ g\right) \otimes \frac{\partial g^k}{\partial x^{\mathbf{i}}}\right)(\underline{x}\ldots) \tag{A15}$$

$$\equiv_{\text{def}} \quad (\underline{x}\ldots) \mapsto \sum_k \left(\frac{\partial f^{\mathbf{j}}}{\partial y^k} \otimes^{(g \times \mathrm{id})} \frac{\partial g^k}{\partial x^{\mathbf{i}}}\right)(\underline{x}\ldots) \tag{A16}$$

$$\equiv_{\text{def}} \quad \bigoplus_k \left(\frac{\partial f^{\mathbf{j}}}{\partial y^k} \otimes^{(g \times \mathrm{id})} \frac{\partial g^k}{\partial x^{\mathbf{i}}}\right) \tag{A17}$$

$$\cong_{\mathrm{T}} \quad \left(\mathrm{J}^{\mathbf{j}}_{k'} \quad \mathrm{J}^{k'}_{\mathbf{i}''}\right) \tag{A18}$$

$$\equiv \quad \mathrm{J}^{\mathbf{j}}_{k'}\mathrm{J}^{k'}_{\mathbf{i}''} \tag{A19}$$

APPENDIX B

$$\frac{\partial\left(f^{\mathbf{j}} \circ g\right)}{\partial x^1} \tag{B1}$$

$$\equiv_{\text{def}} \quad (x^1,x^2) \mapsto \left(\mathbf{z} \mapsto \left(f^{\mathbf{j}} \circ g\right)(\mathbf{z},x^2)\right)'(x^1) \tag{B2}$$

$$\equiv_{\circ} \quad (x^1,x^2) \mapsto \left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(g^1(\mathbf{z},x^2),g^2(\mathbf{z},x^2)\right)\right)'(x^1) \tag{B3}$$

$$=_{\mathrm{lin}'} \quad (x^1,x^2) \mapsto \left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(g^1(\mathbf{z},x^2),g^2(x^1,x^2)\right)\right)'(x^1) + \left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(g^1(x^1,x^2),g^2(\mathbf{z},x^2)\right)\right)'(x^1) \tag{B4}$$

$$\equiv_{\circ} \quad (x^1,x^2) \mapsto \left(\overbrace{\left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(\mathbf{z},g^2(x^1,x^2)\right)\right)}^{a^1} \circ \overbrace{\left(\mathbf{z} \mapsto g^1(\mathbf{z},x^2)\right)}^{b^1}\right)'(x^1) + \left(\overbrace{\left(\mathbf{z} \mapsto f^{\mathbf{j}}\left(g^1(x^1,x^2),\mathbf{z}\right)\right)}^{a^2} \circ \overbrace{\left(\mathbf{z} \mapsto g^2(\mathbf{z},x^2)\right)}^{b^2}\right)'(x^1) \tag{B5}$$

$$\equiv_{\text{def}} \quad (x^1,x^2) \mapsto \left(a^1 \circ b^1\right)'(x^1) + \left(a^2 \circ b^2\right)'(x^1) \tag{B6}$$

$$= \text{chain}' \quad (x^1,x^2) \mapsto \left(\left(a^{1\prime} \circ b^1\right)(x^1)\right) \cdot b^{1\prime}(x^1) + \left(\left(a^{2\prime} \circ b^2\right)(x^1)\right) \cdot b^{2\prime}(x^1) \tag{B7}$$

$$\equiv \text{def} \quad (x^1,x^2) \mapsto \underbrace{\left(\left(\overbrace{(\mathbf{z} \mapsto f^j(\mathbf{z},g^2(x^1,x^2)))'}^{a^{1\prime}} \circ \overbrace{(\mathbf{z} \mapsto g^1(\mathbf{z},x^2))}^{b^1}\right)(x^1)\right)}_{\frac{\partial f^j}{\partial y^1}(g^1(x^1,x^2))} \cdot \overbrace{(\mathbf{z} \mapsto g^1(\mathbf{z},x^2))'(x^1)}^{b^{1\prime}(x^1)}_{\frac{\partial g^1}{\partial x^1}(x^1,x^2)} + \text{ one more term} \tag{B8}$$

$$\equiv \beta \quad (x^1,x^2) \mapsto \left(\left((\mathbf{z} \mapsto f^j(\mathbf{z},g^2(x^1,x^2)))' \circ (\mathbf{z} \mapsto g^1(\mathbf{z},x^2))\right)(x^1)\right) \cdot \underbrace{\left((y^1,y^2) \mapsto (\mathbf{z} \mapsto g^1(\mathbf{z},y^2))'(y^1)\right)}_{\frac{\partial g^1}{\partial x^1}}(x^1,x^2) + \text{ one more term} \tag{B9}$$

$$\equiv \text{def} \quad (x^1,x^2) \mapsto \underbrace{\left(\left((\mathbf{z} \mapsto f^j(\mathbf{z},g^2(x^1,x^2)))' \circ (\mathbf{z} \mapsto g^1(\mathbf{z},x^2))\right)(x^1)\right)}_{\frac{\partial f^j}{\partial y^1}(g^1(x^1,x^2))} \cdot \frac{\partial g^1}{\partial x^1}(x^1,x^2) + \text{ one more term} \tag{B10}$$

$$\equiv \circ \quad (x^1,x^2) \mapsto \underbrace{\left((\mathbf{z} \mapsto f^j(\mathbf{z},g^2(x^1,x^2)))'(g^1(x^1,x^2))\right)}_{\frac{\partial f^j}{\partial y^1}(g^1(x^1,x^2))} \cdot \frac{\partial g^1}{\partial x^1}(x^1,x^2) + \text{ one more term} \tag{B11}$$

$$\equiv \beta \quad (x^1,x^2) \mapsto \underbrace{\left((y^1,y^2) \mapsto (\mathbf{z} \mapsto f^j(\mathbf{z},y^2))'(y^1)\right)}_{\frac{\partial f^j}{\partial y^1}}(g^1(x^1,x^2),g^2(x^1,x^2)) \cdot \frac{\partial g^1}{\partial x^1}(x^1,x^2) + \text{ one more term} \tag{B12}$$

$$\equiv \text{def} \quad (x^1,x^2) \mapsto \frac{\partial f^j}{\partial y^1}(g^1(x^1,x^2),g^2(x^1,x^2)) \cdot \frac{\partial g^1}{\partial x^1}(x^1,x^2) + \frac{\partial f^j}{\partial y^2}(g^1(x^1,x^2),g^2(x^1,x^2)) \cdot \frac{\partial g^2}{\partial x^1}(x^1,x^2) \tag{B13}$$

$$\equiv \circ \quad (x^1,x^2) \mapsto \left(\frac{\partial f^j}{\partial y^1} \circ g\right)(x^1,x^2) \cdot \frac{\partial g^1}{\partial x^1}(x^1,x^2) + \left(\frac{\partial f^j}{\partial y^2} \circ g\right)(x^1,x^2) \cdot \frac{\partial g^2}{\partial x^1}(x^1,x^2) \tag{B14}$$

$$\equiv \text{def} \quad (x^1,x^2) \mapsto \left(\left(\frac{\partial f^j}{\partial y^1} \circ g\right) \otimes \frac{\partial g^1}{\partial x^1}\right)(x^1,x^2) + \left(\left(\frac{\partial f^j}{\partial y^2} \circ g\right) \otimes \frac{\partial g^2}{\partial x^1}\right)(x^1,x^2) \tag{B15}$$

$$\equiv \text{def} \quad (x^1,x^2) \mapsto \left(\frac{\partial f^j}{\partial y^1} \otimes^{(g \times \text{id})} \frac{\partial g^1}{\partial x^1}\right)(x^1,x^2) + \left(\frac{\partial f^j}{\partial y^2} \otimes^{(g \times \text{id})} \frac{\partial g^2}{\partial x^1}\right)(x^1,x^2) \tag{B16}$$

$$\equiv \text{def} \quad \left(\frac{\partial f^j}{\partial y^1} \otimes^{(g \times \text{id})} \frac{\partial g^1}{\partial x^1}\right) \oplus \left(\frac{\partial f^j}{\partial y^2} \otimes^{(g \times \text{id})} \frac{\partial g^2}{\partial x^1}\right) \tag{B17}$$

$$\cong \text{T} \quad \left(\mathrm{J}^j_{1'} \quad \mathrm{J}^{1'}_{1''}\right) +_T \left(\mathrm{J}^j_{2'} \quad \mathrm{J}^{2'}_{1''}\right) \tag{B18}$$

$$\equiv \quad \mathrm{J}^j_{1'}\mathrm{J}^{1'}_{1''} +_T \mathrm{J}^j_{2'}\mathrm{J}^{2'}_{1''} \tag{B19}$$

APPENDIX C

$$\frac{i : (x, a) \in \Gamma}{\Gamma \vdash x \ \because \ i : a} \ (- \because -)$$

$$\frac{\Gamma, (x, \mathsf{tuple}\ m) \vdash T : \mathsf{tuple}\ n}{\Gamma \vdash \lambda^t m\ {}^t x\ .\ T : \mathsf{funMN}\ m\ n} \ (\lambda^t -\ {}^t -\ .\ -)$$

$$\frac{\Gamma, (x, \mathsf{tuple}\ m) \vdash T : \ \mathsf{scalar}}{\Gamma \vdash \lambda^t m\ {}^s x\ .\ T : \mathsf{funM1}\ m} \ (\lambda^t -\ {}^s -\ .\ -)$$

$$\frac{\Gamma, (x, \ \mathsf{scalar}\ ) \vdash T : \mathsf{tuple}\ n}{\Gamma \vdash \lambda^{st} x\ .\ T : \mathsf{fun1N}\ n} \ (\lambda^{st} -\ .\ -)$$

$$\frac{\Gamma, (x, \ \mathsf{scalar}\ ) \vdash T : \ \mathsf{scalar}}{\Gamma \vdash \lambda^{ss} x\ .\ T : \ \mathsf{fun11}} \ (\lambda^{ss} -\ .\ -)$$

$$\frac{\Gamma, (x, \mathsf{index}\ k) \vdash T : \ \mathsf{scalar}}{\Gamma \vdash \Sigma\ k\ {}^s x\ .\ T : \ \mathsf{scalar}} \ (\Sigma -\ {}^s -\ .\ -)$$

$$\frac{\Gamma \vdash T : \ \mathsf{fun11} \qquad \Gamma \vdash U : \ \mathsf{scalar}}{\Gamma \vdash T\ {}^{11}(\!|\ U\ |\!) : \ \mathsf{scalar}} \ (-\ {}^{11}(\!|-|\!))$$

$$\frac{\Gamma \vdash T : \mathsf{funM1}\ m \qquad \Gamma \vdash U : \mathsf{tuple}\ m}{\Gamma \vdash T\ {}^{m1}(\!|\ U\ |\!) : \ \mathsf{scalar}} \ (-\ {}^{m1}(\!|-|\!))$$

$$\frac{\Gamma \vdash T : \mathsf{fun1N}\ n \qquad \Gamma \vdash U : \ \mathsf{scalar}}{\Gamma \vdash T\ {}^{1n}(\!|\ U\ |\!) : \mathsf{tuple}\ n} \ (-\ {}^{1n}(\!|-|\!))$$

$$\frac{\Gamma \vdash T : \mathsf{funMN}\ m\ n \qquad \Gamma \vdash U : \mathsf{tuple}\ m}{\Gamma \vdash T\ {}^{mn}(\!|\ U\ |\!) : \mathsf{tuple}\ n} \ (-\ {}^{mn}(\!|-|\!))$$

$$\frac{\Gamma \vdash T : \mathsf{tuple}\ k \qquad i : \mathbb{N}_k \qquad \Gamma \vdash U : \ \mathsf{scalar}}{\Gamma \vdash T\ {}^{k}[\bullet i := U] : \mathsf{tuple}\ k}$$

$$\frac{\Gamma \vdash T : \mathsf{fun1N}\ n \qquad i : \mathbb{N}_n \qquad \Gamma \vdash U : \ \mathsf{scalar}}{\Gamma \vdash T\ {}^{n}[\bullet i := U] : \mathsf{fun1N}\ n}$$

$$\frac{\Gamma \vdash T : \mathsf{funMN}\ m\ n \qquad i : \mathbb{N}_n \qquad \Gamma \vdash U : \ \mathsf{scalar}}{\Gamma \vdash T\ {}^{*}[\bullet i := U] : \mathsf{funMN}\ m\ n}$$

$$\frac{\Gamma \vdash T : \mathsf{tuple}\ k \qquad \Gamma \vdash U : \mathsf{index}\ k \qquad \Gamma \vdash V : \ \mathsf{scalar}}{\Gamma \vdash T\ {}^{ki}[\bullet U := V] : \mathsf{tuple}\ k}$$

$$\frac{\Gamma \vdash T : \mathsf{fun1N}\ n \qquad \Gamma \vdash U : \mathsf{index}\ n \qquad \Gamma \vdash V : \ \mathsf{scalar}}{\Gamma \vdash T\ {}^{ni}[\bullet U := V] : \mathsf{fun1N}\ n}$$

$$\frac{\Gamma \vdash T : \mathsf{funMN}\ m\ n \qquad \Gamma \vdash U : \mathsf{index}\ n \qquad \Gamma \vdash V : \ \mathsf{scalar}}{\Gamma \vdash T\ {}^{*i}[\bullet U := V] : \mathsf{funMN}\ m\ n}$$

$$\frac{\Gamma \vdash T : \mathsf{tuple}\ k \qquad i : \mathbb{N}_k}{\Gamma \vdash T \hat{}^k i : \ \mathsf{scalar}} \ (-\hat{}^k -)$$

$$\frac{\Gamma \vdash T : \mathsf{fun1N}\ n \qquad i : \mathbb{N}_n}{\Gamma \vdash T \hat{}^n i : \ \mathsf{fun11}} \ (-\hat{}^n -)$$

$$\frac{\Gamma \vdash T : \mathsf{funMN}\ m\ n \qquad i : \mathbb{N}_n}{\Gamma \vdash T \hat{}^* i : \mathsf{funM1}\ m} \ (-\hat{}^* -)$$

$$\frac{\Gamma \vdash T : \mathsf{tuple}\ k \qquad \Gamma \vdash U : \mathsf{index}\ k}{\Gamma \vdash T \hat{}^{ki} U : \ \mathsf{scalar}} \ (-\hat{}^{ki} -)$$

$$\frac{\Gamma \vdash T : \mathsf{fun1N}\ n \qquad \Gamma \vdash U : \mathsf{index}\ n}{\Gamma \vdash T \hat{}^{ni} U : \ \mathsf{fun11}} \ (-\hat{}^{ni} -)$$

$$\frac{\Gamma \vdash T : \mathsf{funMN}\ m\ n \qquad \Gamma \vdash U : \mathsf{index}\ n}{\Gamma \vdash T \hat{}^{*i} U : \mathsf{funM1}\ m} \ (-\hat{}^{*i} -)$$

$$\frac{\Gamma \vdash T : \ \mathsf{fun11} \qquad \Gamma \vdash U : \ \mathsf{fun11}}{\Gamma \vdash T \circ^{111} U : \ \mathsf{fun11}} \ (- \circ^{111} -)$$

$$\frac{\Gamma \vdash T : \mathsf{funM1}\ k \qquad \Gamma \vdash U : \mathsf{fun1N}\ k}{\Gamma \vdash T \circ^{1k1} U : \ \mathsf{fun11}} \ (- \circ^{1k1} -)$$

$$\frac{\Gamma \vdash T : \mathsf{funMN}\ kn \qquad \Gamma \vdash U : \mathsf{funMN}\ mk}{\Gamma \vdash T \circ^{mkn} U : \mathsf{funMN}\ m\ n} \ (- \circ^{mkn} -)$$

$$\frac{\Gamma \vdash T : \mathsf{fun1N}\ n \qquad \Gamma \vdash U : \mathsf{funM1}\ m}{\Gamma \vdash T \circ^{m1n} U : \mathsf{funMN}\ m\ n} \ (- \circ^{m1n} -)$$

$$\frac{\Gamma \vdash T : \ \mathsf{fun11}}{\Gamma \vdash T\ {}'^s : \ \mathsf{fun11}} \ (-\ {}'^s)$$

$$\frac{\Gamma \vdash T : \ \mathsf{scalar} \qquad \Gamma \vdash U : \ \mathsf{scalar}}{\Gamma \vdash T \cdot^s U : \ \mathsf{scalar}} \ (- \cdot^s -)$$